HUMPHRIES

ISSN 0265-2919

90p 71

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

A PATTERN EMERGES We provide a pattern recognition program that uses many of the ideas discussed last week — 1401

HARDWARE

DRAWING COMPARISONS The performance of a supercomputer like the Cray-1 provides a telling comparison with a Z80-based micro — 1404

MOUSE PIECE The Magic Mouse from SMC Supplies is a handy add-on to a Commodore 64 system — 1410

COMPUTER SCIENCE

ACTIVE LIST To conclude our LISP series, we show how the language is used to tackle a chess-based problem — 1415

JARGON

FROM QUERY LANGUAGE TO RAM A weekly glossary of computing terms — 1409

PROGRAMMING PROJECTS

DO NOT PASS GO! We give the initial set-up routine of the Go game for three more popular home micros — 1406

MACHINE CODE

DRIVE CONTROL A look at the hook codes available on the Sinclair Microdrive — 1418

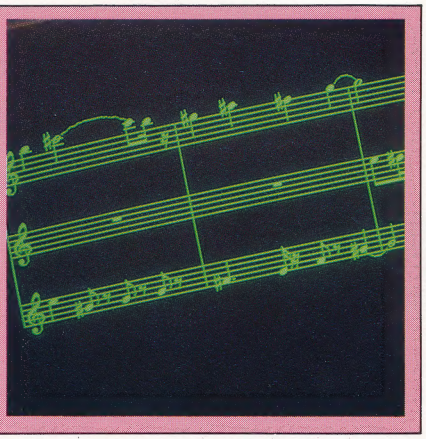
WORKSHOP

HIGH FIDELITY We provide a Digital Recorder program for the MIDI interface — 1412

REFERENCE CARD More of the Commodore 64's memory map — INSIDE BACK COVER

Next Week

●Digital Research's Dr LOGO is now implemented on a wide range of machines, from the Amstrad CPC 464 to the IBM PC. We take a look at the various versions available.
●One of the primary areas of research in artificial intelligence is the creation of 'intelligent' robots. We separate the science fact from the science fantasy.
●Our MIDI interface project bows out with a look at some useful software techniques.



QUIZ

1) What is the purpose of the 'front end – interface' on the Cray-1 supercomputer?
2) What is the difference between 'step-time' and 'real-time' music programming?
3) What function is performed by the Magic Mouse Controller program?

Answers To Last Week's Quiz

1) In LISP, a list is a collection of data items that can be acted upon, whereas a function is a command that performs a specific action.
2) The MSB of the status byte is set to one. This indicates that a fresh message is arriving.
3) A controlled hallucination occurs when the computer projects a template from memory onto the picture that it is 'seeing' to discover whether they match.
4) A hook code is a byte value used to select a specific routine within the Spectrum's shadow ROM.

New World Simulation Game Amendment

Spectrum owners should renumber the final three modules of the New World simulation game (on page 1356) by reducing all line numbers above 10000 by 2000, so that these modules relocate between lines 8000 and 8573. Calls to these modules should be changed as follows:

891 GOSUB 8000
892 GOSUB 8070
893 GOSUB 8300
894 GOSUB 8500

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editor** Bobby Pickering; **Designers** Julian Dorr, Robin Allen; **Staff Writer** Steve Malone; **Art Assistant** Caroline Clayton, **Sub Editor** Jon Kaye; **Contributors** Richard Forsyth, Marcus Jeffery, Martin Young, Joe Pritchard, Steve Malone, Tony Samuels, Steve Colwill, David Mudd; **Software Consultants** Pilot Software.City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Alastair Gourlay; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: Aus$2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZ$2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY CRISPIN THOMAS ILLUSTRATION BY JOHN CLEMENTSON

TONY SLEEP AT COMPUTER RECOGNITION SYSTEM

What's That Expression On Your Face?
Igor Aleksander's WISARD system, developed at Imperial College, London, uses a 512 by 512 grid and a megabyte of RAM in order to recognise patterns displayed on a television screen. It is so sensitive that it can be taught to discriminate between smiling and frowning faces

A PATTERN EMERGES

We continue our investigation into computer vision systems, looking at a method of scanning patterns for significant features. We also develop a BASIC program that demonstrates the pattern recognition methods used by the WISARD system.

So-called 'bottom-up' pattern recognition systems attempt to extract significant features from a given image and reduce it to a simpler, more highly abstracted pattern. One method of finding such features within a pattern is to use local operators to scan it. Each local operator scans a small area at a time and multiplies the grey intensities of the points under the operator by weighting factors. These factors are designed to produce high scores when the sought-after feature is located. By using a number of different local operators, the positions of important features, such as horizontal, vertical and diagonal lines or patches of light and dark, can be extracted.

Igor Aleksander's WISARD program, described in the previous instalment (see page 1381), uses the principle of 'memory address generation' from a pattern in order to remember the pattern and recognise it later.

The great advantage of WISARD is that all the RAM-banks are addressed in parallel. Thus it can work at high speed with sharply defined television pictures. Our example program in BASIC simulates such a system sequentially. It is much slower, but it amply demonstrates the principles involved.

The system uses sextuples for its discriminators (these are less powerful than octuples but save space), and it requires 20,480 (40×64×8) bits of RAM, or 2,560 bytes. This is because there are 40 sextuples, each of which can be in 64 states and can thus address a RAM-bank of 64 locations. Each location contains eight bits, which permits the system to discriminate between eight different classes of input.

The program has two phases: the first is a training stage in which the computer can be given examples for each different class of input. The second phase involves recognition of a pattern by comparison with previously learned examples.

For instance, suppose that during the training phase, sextuple number 20 gives the response 110010 (50) when pattern class 4 is present. In this case, the fourth bit in location 50 of RAM-bank 20 will be set to 1. If, during the recognition phase, the

discriminator responds with a 50 again, it will find the fourth bit set in the address specified, tending to confirm the presence of pattern number 4.

A series of bytes, starting at D, are used to hold the pattern data, and the entire array will be cleared at the start of each run. This is achieved by subroutine 1000. Subroutine 2000 allows you to 'paint' simple 16 by 16 images on the screen, by using the keys U, D, L and R to move, and using * and the Space bar to create the pattern.

The program can be modified so that you can load and save the array D to disk or tape. As it stands, you have to teach it a classification task from scratch each time you run the program, so feel free to make any necessary improvements in this respect. Ideally, of course, the system should be linked to the outside world through an analogue-to-digital converter attached to a camera of some kind — for an example of this sort of system, see page 749.

Note that the assignment of pixels to sextuples is random — but it must be repeatable.

The pattern image is initially held in a two-dimensional array, I(,), using one to represent a star and zero to represent a space. During the training phase, addresses are generated from randomly selected elements of I(,). The appropriate bit (depending on the class being learned) is set at line 210. During the recognition phase, the same sequence of randomly selected addresses must be generated in order to compare the pattern being tested with the classifications learned earlier.

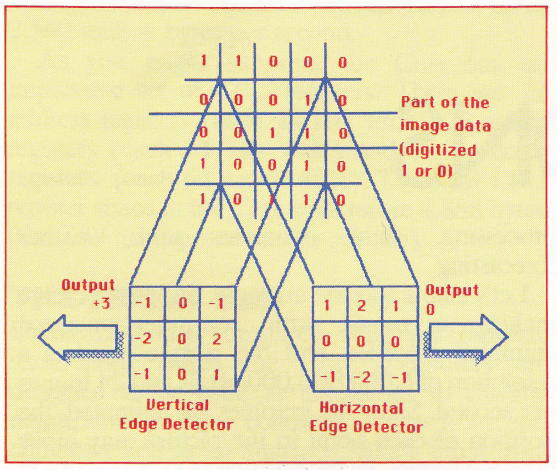
Pattern Recogniser Program

BBC Micro:

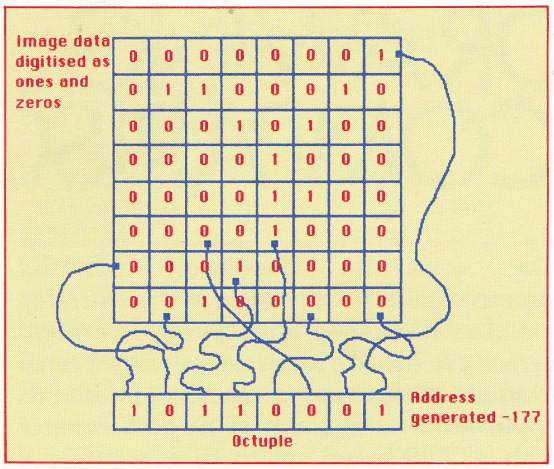
```
  10 REM ************************
  30 REM ** BBC WISARD-TYPE     **
  40 REM ** PATTERN RECOGNIZER **
  50 REM ************************
  55 MODE 7
  56 SX%=40: DS%=6: RB%=2^DS%
  57 MEMO%=SX%*RB%
  60 DIM I%(16,16), CS%(7)
  64 DIM D% MEMO%
  65 @%=4 : REM o/p format
  66 REM D% is a byte-array (BBC)
  70 REM -- I% is image array, D% is address array.
  80 REM -- Note: I% contains either 0 or 1
  85 PRINT "Total RAM size = ";MEMO%;" bytes."
  88 GOSUB 1000: REM clear D% array.
  90 REM -- First the Training Phase:
  92 MOOD$="Training"
  95 INPUT "Which class is it (0..7) ",C%
  96 IF C%<0 OR C%>7 THEN GOTO 90
 100 FOR I%=1 TO 16: FOR J%=1 TO 16: I%(I%,J%)=0
 101 NEXT: NEXT
 110 REM -- Now get user-painted image:
 120 GOSUB 2000
 130 R=RND(-1): REM random seed value
 140 FOR I%=1 TO SX%
 150 A%=RB% * (I%-1)
 155 REM A% is base address of RAM-bank.
 160 FOR J%=0 TO (DS%-1)
 170 R1%=INT(RND(1)*16 + 1)
 180 R2%=INT(RND(1)*16 + 1)
 190 A%=A%+ I%(R1%,R2%) * 2^J%
 200 NEXT J%
 202 REM R1% and R2% are 'random' co-ords.
 210 PROCdset(A%,C%)
 220 NEXT I%
 230 INPUT "Another training session (Y/N) ", A$:
IF A$="Y" THEN GOTO 90
 233 IF A$="y" THEN GOTO 90
 235 MOOD$="Recognition"
 236 REM -- Now the recognition phase:
 240 FOR I%=1 TO 16: FOR J%=1 TO 16
 244 I%(I%,J%)=0
 245 NEXT: NEXT
 250 REM -- Get an image for classification:
 260 GOSUB 2000
 270 R=RND(-1)
 280 FOR C%=0 TO 7: CS%(C%)=0: NEXT
 290 FOR I%=1 TO SX%
 300 A%=RB% * (I%-1)
 310 FOR J%=0 TO DS%-1
 320 R1%=INT(RND(1)*16 + 1)
 330 R2%=INT(RND(1)*16 + 1)
 340 A%=A%+I%(R1%,R2%) * 2^J%
 350 NEXT
 355 FOR C%=0 TO 7
 360 IF FNdget(A%,C%) > 0 THEN CS%(C%)=CS%(C%)+1
 370 NEXT C%
 380 NEXT I%
 382 CX%=0
 385 FOR C%=0 TO 7
 388 PRINT "Class";C%;" has a score of ";CS%(C%)/
SX%*100;SPC(8)
 390 IF CS%(C%)>CS%(CX%) THEN CX%=C%
 400 NEXT C%
 404 PRINT "Most likely class is no. ";CX%
 410 INPUT "Do you want to classify another image
(Y=Yes) ", A$
 420 IF A$="Y" OR A$="y" THEN GOTO 240
 440 PRINT: PRINT "Bye!"
 444 END
 999 :
1000 REM -- Memory Initialization:
1010 FOR I%=0 TO MEMO%
1020 REM -- uses byte-subscripting (?):
1030 ?(D%+I%)=0
1040 NEXT
1050 RETURN
1199 :
1200 DEF PROCdset(A%,C%)
1210 REM --------
1220 ?(D%+A%) = (D%?A%) OR 2^C%
1230 ENDPROC
1240 REM sets a bit in D% array
1244 :
1250 DEF FNdget(A%,C%)
1260 REM ------
1270 =(D%?A%) AND 2^C%
1280 REM extracts bit from D% array
1288 REM A% is address; C% is 0..7
1290 :
2000 REM -- Image-making Routine:
2010 CLS: PRINT TAB(0,20);"(Use: U, D, L, R, X, *
or space"
2012 PRINT "to define the image.)"
2013 PRINT MOOD$;" phase."
2015 PRINT TAB(1,1);
2020 T%=0: A$=" "
2030 H%=1: V%=1
2040 REPEAT  C$=INKEY$(222)
2050 IF C$="U" THEN V%=V%-1
2060 IF C$="D" THEN V%=V%+1
2070 IF C$="L" THEN H%=H%-1
2080 IF C$="R" THEN H%=H%+1
2090 IF H%>16 THEN H%=1
2095 IF H%<1 THEN H%=16
2100 IF V%>16 THEN V%=1
2105 IF V%<1 THEN V%=16
2110 IF C$=" " THEN T%=0: A$=C$
2120 IF C$="*" THEN T%=1: A$=C$
2130 I%(H%,V%)=T%
2140 PRINT TAB(H%,V%);A$;
2150 UNTIL C$="X"
2155 GOSUB 2200 : REM Re-display
2160 RETURN
2170 :
2200 REM -- display Routine:
2210 FOR H%=1 TO 16
2220 FOR V%=1 TO 16
2230 IF I%(H%,V%)>0 THEN PRINT TAB(H%,V%);":" ELS
E PRINT TAB(H%,V%);" ";
2240 NEXT: NEXT
2250 PRINT TAB(0,17);
2260 RETURN
2270 :
2300 REM 'painting' commands are:
2301 REM U & D for Up and Down;
2302 REM L & R for Left and Right;
2303 REM * & space for On and Off;
2304 REM X to exit (image done).
```

Commodore 64:

```
  10 REM **** CBM 64 PATTERN ****
  20 REM ****   RECOGNISER   ****
  30 FORI=1 TO 25:DW$=DW$+CHR$(17):NEXT I
  56 SX=40:DS=6:RB=2^DS
  57 ME=SX*RB
  60 DIM I(16,16),C(8)
  64 D=12*4096:REM USE $C000- FOR D ARRAY
  85 PRINT"TOTAL RAM SIZE=";ME;" BYTES"
  88 GOSUB 1000:REM CLEAR D ARRAY
  90 REM **** THE TRAINING PHASE ****
  92 M$="TRAINING"
  95 INPUT"WHICH CLASS IS (1...8)";C
  96 IFC<1 OR C>8 THEN 95
 100 FOR I=1 TO 16:FOR J=1 TO 16:I(I,J)=0
 101 NEXT J:NEXT I
 110 REM **** NOW GET USERPAINTED IMAGE ****
 120 GOSUB 2000:REM IMAGE MAKER
```



CAROLINE CI AYTON ON THE MACINTOSH



Octuple Eye
Groups of eight pixels are arbitrarily connected to eight bit registers to form the basis of WISARD's pattern recognition system. Each octuple corresponds to a bank of 256 memory locations. During the training phase the value in the the octuple forms an address within its RAM bank dependent on the pixel pattern and a 1 is written to this address. If the same pattern is present during the recognition phase then the same RAM bank address will be generated and the 1 present will signify recognition

Local Operators
An example of the kind of low-level processing that might be employed in a bottom-up pattern recognition system is illustrated on the far-left. The three by three grids are so-called 'local operators', which scan over the image data and record high scores at edge transitions, thus giving a crude outline of the picture in terms of straight lines. The operators are moved across the image data and each number is multiplied by the grey-level intensity of the pattern section over which it is placed. Adding products gives high scores when the desired feature is present, low scores otherwise. Each of the nine numbers in a local operator is a weighting factor; the pattern of these weightings determines what features will be detected

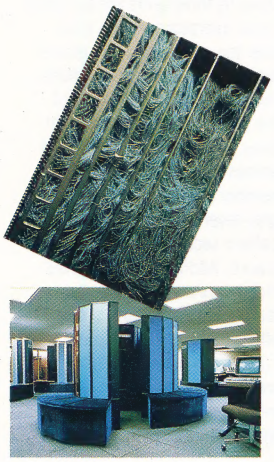
```
130 R=RND(-1):REM RANDOM SEED VALUE
140 FORI=1 TO SX
150 A=RB*(I-1):REM CALC ADDRESS OF RAM BANK
160 FOR J=0 TO (DS-1)
170 R1=INT(RND(1)*16+1):R2=INT(RND(1)*16+1)
190 A=A+I(R1,R2)*2^J
200 NEXT J
210 POKE(D+A),(PEEK(D+A) OR 2^(C-1))
220 NEXT I
230 INPUT"ANOTHER TRAINING SESSION (Y/N)";A$
233 IF A$="Y" THEN 90:REM REPEAT
234 REM **** NOW RECOGINITION PHASE ****
235 M$="RECOGNITION"
240 FOR I=1 TO 16:FOR J=1 TO 16
244 I(I,J)=0
245 NEXT J:NEXT I
250 GOSUB 2000:REM GET AN IMAGE FOR CLASSIFICATION
270 R=RND(-1)
280 FORC=1 TO 8:C(C)=0:NEXT C
290 FOR I=1 TO SX
300 A=RB*(I-1)
310 FOR J=0 TO DS-1
320 R1=INT(RND(1)*16+1):R2=INT(RND(1)*16+1)
340 A=A+I(R1,R2)*2^J
350 NEXT J
355 FOR C=1 TO 8
360 IF (PEEK(D+A)AND 2^(C-1))>0 THEN C(C)=C(C)+1
370 NEXT C
380 NEXT I
381 PRINTCHR$(147)
382 CX=0:X=0:Y=17:GOSUB 3000
385 FOR C=1 TO 8
388 PRINT"CLASS"C"HAS A SCORE OF";C(C)/SX*100
390 IF C(C)>C(CX) THEN CX=C
400 NEXT C
404 PRINT"MOST LIKELY CLASS IS NO.";CX
410 INPUT "CLASSIFY ANOTHER IMAGE (Y/N)";A$
420 IF A$="Y" THEN 240:REM REPEAT
444 END
999 :
1000 REM ***** INIT MEMORY ****
1010 FOR I=0 TO ME:POKE(D+I),0:NEXT I
1030 RETURN
1040 :
2000 REM **** IMAGE MAKER ****
2005 PRINTCHR$(147)
2010 X=0:Y=20:GOSUB 3000:PRINT"USE U,D,L,R,X,* OR
SPACE"
2012 PRINT"TO DEFINE THE IMAGE"
2013 PRINTM$;" PHASE"
2015 X=1:Y=1:GOSUB 3000
2020 T=0:A$=" ":H=1:V=1
2040 GET C$
2050 IF C$="U" THEN V=V-1:IF V<1 THEN V=16
2060 IF C$="D" THEN V=V+1:IF V>16 THEN V=1
2070 IF C$="L" THEN H=H-1:IF H<1 THEN H=16
2080 IF C$="R" THEN H=H+1:IF H>16 THEN H=1
2110 IF C$=" " THEN T=0:A$=C$
2120 IF C$="*" THEN T=1:A$=C$
2130 I(H,V)=T
2140 X=H:Y=V:GOSUB 3000:PRINTA$;
2150 IF C$<>"X" THEN 2040
2155 GOSUB 2200
2160 RETURN
2170 :
2200 REM **** DISPLAY ROUTINE ****
2210 FOR H=1 TO 16
2220 FOR V=1 TO 16
2230 X=H:Y=V:GOSUB3000:IF I(H,V)=0 THEN PRINT":":G
OTO2240
2235 PRINT" ";
2240 NEXT V:NEXT H
2250 X=0:Y=17:GOSUB 3000
2260 RETURN
2270 :
3000 REM **** TAB COMMAND ****
3010 PRINTCHR$(19);TAB(X);LEFT$(DW$,Y);
3020 RETURN
```

Basic Flavours

Spectrum:
The Spectrum does not support bitwise AND and OR instructions. We therefore have to write short machine code routines to perform these operations. These are given in the form of a BASIC Loader. Make the following alterations to the Commodore 64 version given:

```
30 CLEAR 49999:GOSUB 4000

64 LET D=50018

130 RANDOMIZE 1

210 LET X1=PEEK(D+A):LET Y1=2↑(C-1)
    :GOSUB 6000
215 POKE(D+A),R1

270 RANDOMIZE 1

360 LET X1=PEEK(D+A) :LET Y1=2↑(C-1)
    :GOSUB 5000
365 IF R1>0 THEN C(C)=C(C)+1

381  CLS

2005 CLS

3010 PRINT AT Y,X;

4000 FOR I=50000 TO 50017
4010 READ A:POKE I,A:NEXT I
4020 RETURN
4030 DATA 62,0,14,0,161,6,0,79,201
4040 DATA 62,0,14,0,177,6,0,79,201

5000 REM **** AND ****
5010 POKE 50001,X1:POKE 50003,Y1
5020 LET R1=USR 50000:RETURN

6000 REM **** OR ****
6010 POKE 50010,X1:POKE 50012,Y1
6020 LET R1=USR 50009:RETURN
```

DRAWING COMPARISONS



The Favourite's Form
The Cray-1 (pictured above) offers the user enormous processing power under the control of a front-end computer such as an IBM, DEC, or similar. The architecture of the Cray-1 (shown below) features 32 Mbytes of main memory and 4,888 bytes of register space, contained within a system boasting 3,400 separate printed circuit boards and over 90 kilometres of wire connecting them

The need for immensely powerful supercomputers capable of performing 'number crunching' applications is evident in many fields. We focus on the Cray-1 as an example of this type of machine, look at its applications, and compare its performance with a Z80-based micro in generating a short animated film.

The 'power' of a computer, in crude terms, is the product of its word size, data transfer speed, main memory size and central processing unit machine cycle speed. In microcomputers, the main functions of the traditional CPU are packed into a single microprocessor. These microprocessors include the familiar Z80, the Intel 8088 and 8086 and the Motorola M68000. All use MOS (metal oxide semiconductor) technology for the logic circuits and on-chip memory. Data is transferred and processed in parallel either eight bits or 16 bits at a time. Clock frequencies in microprocessors range from 1MHz to 12MHz.

Specifications such as these, impressive though they are, make microprocessor-based computers unable to cope with the colossal amount of data processing required in applications such as image processing, fluid dynamics and weather forecasting.

Let's take a specific example. Suppose you are making a movie with computer-generated animated graphics, and the picture requires a resolution of 6,000 by 6,000 pixels and 24 frames per second. Since the graphics are animated, the position of each point in the picture may move between frames, so the position of every point will have to be calculated for each frame. That translates into 864 million calculations per second, and each calculation is likely to be quite complex, involving dozens or hundreds of machine code instructions. This adds up to billions of instructions per second. The Time Trials box shows an approximate comparison of the time taken to produce a 10-minute film sequence on a supercomputer and a Z80-based micro.

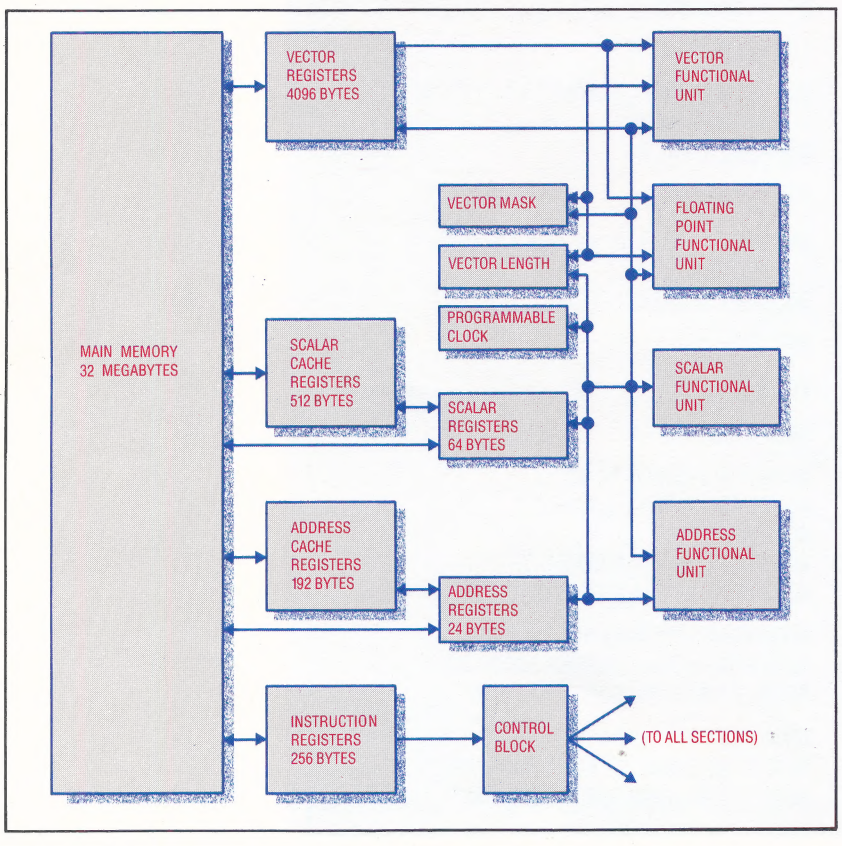
Supercomputers, as the largest mainframes tend to be called, work in much the same way as microcomputers; instructions are fetched from memory, data is fetched from memory, the data is manipulated by the processor according to its instructions and the results are stored in memory. The chief difference is the scale and speed at which these operations are carried out, and the way that the parts of the computers are constructed.

We'll illustrate a 'typical' supercomputer by referring to the Cray-1 S/4400, manufactured by Cray Research. The first Cray-1 was installed in 1976, only four years after the founding of the company. Since then it has established itself as the most famous (and one of the most popular) of the super mainframes.

The Cray-1 CPU fills a six-and-a-half foot high, semicircular cabinet, shaped somewhat like a curved sofa (the cooling system and power supplies are housed under the 'seats'). It gets its speed by using bipolar semiconductor logic (bipolar semiconductors are 'ordinary' transistors as opposed to MOS, CMOS, NMOS, FET — field effect transistor — and other types of semiconductor). The bipolar logic and memory comes in over 200,000 integrated circuits on 3,400 separate printed circuit boards, with over 90 kilometres of wire used for interconnecting them.

The word size used for computations is 64 bits wide (eight times as much data is processed at a time than in an eight-bit Z80) and the system clock speed is 80MHz (80 million cycles per second). A typical 64-bit addition takes just 37.5 nanoseconds. An eight-bit addition in a Z80 running at 4MHz (for example, `ADD A, n`) takes 1.75 microseconds.

The main memory in the Cray is an integral part of the CPU. It contains 32 megabytes organised as



4,194,304 words. The memory can transfer up to 2,560 million bytes per second.

As you might suppose, the Cray has an impressive set of CPU registers. There are 72 address registers (each 24 bits wide), 72 scalar registers (each 64 bits wide) and eight vector registers (each 64 words wide). The total CPU register space in the Cray is therefore 4,888 bytes (the Z80 microprocessor has just 26).

THE FRONT END

Unlike most other computers, which are complete stand-alone units, the Cray computers are designed to be used as extensions to existing mainframe or mini computers. The front end computer acts as an interface, taking input from terminals or card readers, and feeding output to peripherals such as printers or magnetic tape storage units.

Between the Cray CPU and the mainframe front-end computer lies the Cray I/O subsystem, designed to facilitate the high throughput demands of the CPU, and a front-end interface, designed to match the Cray system to the specific characteristics of the IBM, DEC, Data General or other front-end computers. The Cray I/O subsystem consists of from two to four I/O processors, each of which is a powerful minicomputer in its own right.

This description of the Cray-1's architecture barely scratches the surface, though the block diagram of the CPU may help to give you an idea of its sophistication. There is no space to go into the instruction set, the 13 so-called functional units that can operate in parallel, or the vector processing that allows up to 64 pairs of operands to be operated on by a single instruction. Suffice it to say that the Cray is a *very* powerful machine.

Supercomputer Applications

Weather Forecasting
Weather forecasts are now the results of world-wide meteorological data collection, satellite communication and computer modelling. A computer 'model' of the world's weather involves hundreds of millions of calculations on vast arrays of data, with predictions needed in hours. Only immensely powerful computers such as the Cray can handle data in this quantity quickly enough

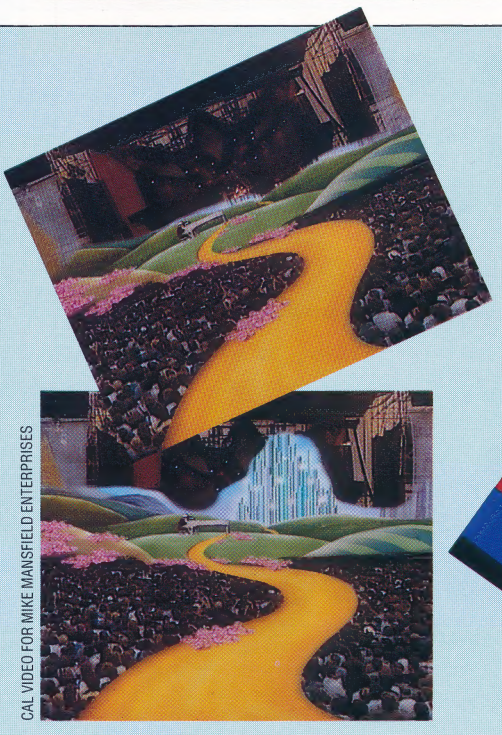
Fluid Dynamics
Fluid dynamics play an important part in the design of fuel-efficient cars, the cooling systems in nuclear power stations, aeronautical engineering and in many other applications. Analysing and predicting the movement of fluids requires reams of data to be analysed. The problems become immense if real-time results are needed. The trouble is that each particle in a fluid — and there are countless trillions of them — will have an effect on all the other particles in the system when it moves. Calculating the behaviour of an entire fluid system therefore calls for massive computing power

Economic Forecasting
Models of the economy are notoriously difficult to construct. As in fluid dynamics, a small change in any part of the system can have effects that ripple through the rest of the system. Attempting to create an economic model of the entire world is hardly practical, but even simplified models are of mind-numbing complexity. Again, extremely fast and powerful computers are needed if analysts are not to be kept months waiting for results. When time is money and speed is of the essence, a supercomputer can seem a viable solution

Time Trials

To compare the processing power of a Cray-1 supercomputer and a Z80-based home computer such as the Amstrad CPC 464, consider the problems of making a ten-minute film sequence of animated, high resolution graphics. We'll assume a picture resolution of 6,000 by 6,000 pixels, and 24 frames per second. We'll also assume that each pixel needs to be individually calculated for each new frame, and that 100 machine code instructions will be needed for the Z80 version, with an average execution time of 19 clock cycles (4.75 microseconds). For the Cray-1, with its powerful vector processing instructions, we'll say that 25 machine code instructions are needed (to err on the conservative side) with an average execution time of four clock cycles (50 nanoseconds).

A Z80 would take $6000^2 \times 24 \times 60 \times 10 \times 100 \times 4.75 \times 10^{-6} = 2.4624 \times 10^8$ seconds or 7.8 years. A Cray-1 would take $6000^2 \times 24 \times 60 \times 10 \times 25 \times 50 \times 10^{-9} = 6.48 \times 10^5$ seconds or 7.5 days



CAL VIDEO FOR MIKE MANSFIELD ENTERPRISES

Visual Revolution
From Elton John's Goodbye Yellow Brick Road to Gillette advertisements, computer-generated graphics are revolutionising video and film production. Images like these can easily be created using modern minicomputer and micro technology. On a larger scale, a Cray-1 was used to generate over 20 minutes of footage for the film The Last Starfighter, performing calculations that would keep an 8-bit micro occupied for over 15 years

CAL VIDEO FOR B.B.D.O.

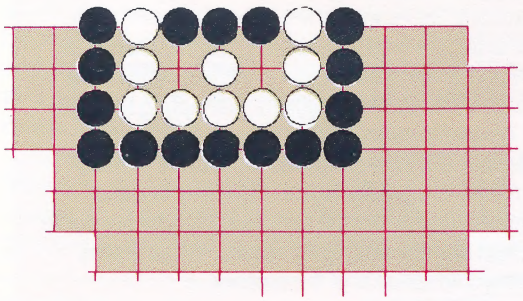
DO NOT PASS GO!

We continue our Go programming project by listing the game's I/O routines for the Commodore 64, Sinclair Spectrum and Amstrad CPC 464/664 micros. These routines create the board display for the game and provide general-purpose subroutines for printing messages and accepting input.

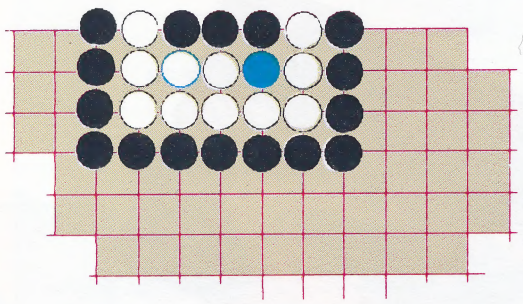
As far as possible, the listings given here form the equivalent program segment to those given on page 1394 for the BBC Micro, and much of the description given in the previous instalment's listings applies equally to the Commodore 64, Sinclair Spectrum and Amstrad CPC 464/664 versions of Go.

To keep things simple, the same line numbers are used in all versions of the program; only where the differences in machine architecture and BASIC dialect become crucial are the original algorithms altered. The main differences between the BBC Micro version and the versions given here, apart from the way in which the four machines handle their screen displays, is that the Amstrad, Spectrum and Commodore 64 versions of BASIC lack procedures and multi-line functions. However, we have simulated these features by using subroutines.

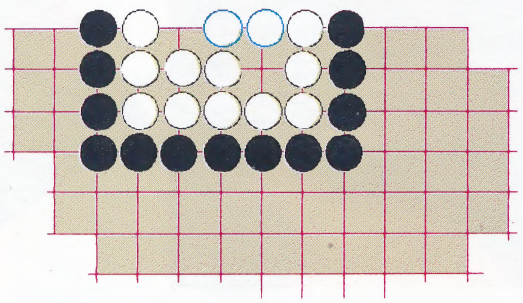
SEKI



Who Surrounds Who?
The white group is completely surrounded by the larger black group and has only two remaining liberties. However, this group also surrounds a smaller black group of three stones



Fatal Mistake
If black plays onto one of white's two remaining liberties, in an attempt to capture the white group, then white can play onto the remaining liberty. As these two liberties are shared by white and the smaller black group, then this black group can be removed from the board



Safety
The white group can now make itself safe by forming two eyes (see page 1366), playing in one of the two positions shown. Black should have left the two original shared liberties vacant — a situation known in Go as SEKI

CAROLINE CLAYTON

Module One

Commodore 64:

```
10 POKE 56578,PEEK(56578)OR3:POKE 56576,
(PEEK(56576)AND252)OR1
15 POKE648,132:POKE 52,128:POKE 56,128
20 DIM SK%(300)
30 POKE 53280,0:POKE 53281,0:PRINT"[CLEA
R][DOWN][DOWN][DOWN][DOWN][DOWN][DOWN][D
OWN][DOWN][DOWN][DOWN]              [WHITE]
PLEASE WAIT":GOSUB 170
40 GOSUB 1270
50 PRINT"[CLEAR]":GOSUB 1730
60 MOVE%=MOVE%+1:REM WHITE MOVES HERE
70 IF FIN% GOTO 100
80 MOVE%=MOVE%+1:REM BLACK MOVES HERE
90 IF NOT FIN% GOTO 60
100 IP%=22:IM%=9:IW%=1:GOSUB 1990
110 IF A$="Y" GOTO 40
115 IF A$<>"N" GOTO 100
120 MP%=24:MM%=5:O$="":GOSUB 2160
130 GOTO130
170 REM INITIALISE ROUTINE
190 BLACK%=1:WHITE%=2:COLOUR%=3
200 MARKER%=4:LIBERTY%=8
220 BOARD=49152
240 DIM CAPTURE%(2)
250 AXIS$="[c 3]ABCDEFGHIJKLMNO"
260 DIM DIR%(4)
270 FOR L=1 TO 4
280 READ DIR%(L)
290 NEXT
300 DATA 16,1,-16,-1
310 GOSUB 390
350 RETURN
390 REM READ-MESSAGES ROUTINE
420 DIM MESS$(9)
430 FOR M=0 TO 9
440 READ MESS$(M)
450 NEXT
460 DATA "O.K. THINKING..."
470 DATA "ILLEGAL ENTRY: "
480 DATA "STONE ALREADY ON POINT: "
490 DATA "ILLEGAL. KO ON POINT: "
500 DATA "ILLEGAL SUICIDE ON POINT: "
510 DATA "O.K. GAME OVER. PRESS RUN/STOP
"
520 DATA ""
530 DATA "  HOW MANY HANDICAP STONES MAY
 I          HAVE (2-9) ?"
```

```
540 DATA "TYPE YOUR MOVE (EG. H8). PASS
OR        QUIT :"
550 DATA "DO YOU WANT TO PLAY AGAIN (Y/N
)?"
560 RETURN
1270 REM INTRODUCTION ROUTINE
1280 GOSUB 1340
1290 GOSUB 1450
1340 REM GAME-INIT ROUTINE
1345 STACK%=1
1360 A1$="      ":A2$="      "
1370 LOCAT%=0:MOVE%=1
1380 FIN%=0
1390 CAPTURE%(1)=0:CAPTURE%(2)=0
1410 RETURN
1450 REM TITLE-SCREEN ROUTINE
1460 POKE 53280,0:POKE 53281,0
1470 PRINT"[CLEAR][DOWN][DOWN][DOWN][DOW
N]                [c 3][RVSON][c V][c I][c
 C][RVSOFF]  [RVSON][c V][c I][c C]"
1480 PRINT"                  [RVSON] [RVSOFF
][c D][c I]  [RVSON] [RVSOFF] [RVSON] "
1490 PRINT"                  [RVSON] [RVSOFF
] [RVSON] [RVSOFF]  [RVSON] [RVSOFF] [RV
SON] "
1500 PRINT"                  [c C][RVSON][c
 I][RVSOFF][c V]  [c C][RVSON][c I][RVSOF
F][c V]"
1510 PRINT"[DOWN][DOWN]          [CYAN]BY
 MARCUS  JEFFERY"
1520 PRINT"[DOWN][PURPLE]YOU WILL PLAY T
HE [WHITE]WHITE [PURPLE]STONES,  AND"
1530 PRINT"THE  COMPUTER  (BEING WEAKER!
)  WILL"
1540 PRINT"HAVE THE [c 3]RED [PURPLE]STO
NES WITH A  HANDICAP"
1550 PRINT"              ADVANTAGE."
1560 IP%=20:IM%=7:IW%=1:GOSUB 1990
1565 HND%=VAL(A$)
1570 IF HND%<2 OR HND%>9 GOTO 1560
1590 RETURN
1730 REM PRINT-BOARD ROUTINE
1750 PRINT"[HOME][YELLOW]       STONES CA
PTURED BY:";
1760 PRINT"[WHITE]       MOVE"
1770 PRINT"[YELLOW]     WHITE =[CYAN] ";C
APTURE%(2);
1780 PRINT TAB(18);"[YELLOW]BLACK =[CYAN
] ";CAPTURE%(1);
1790 PRINT TAB(31);"[WHITE]";MOVE%
1800 PRINT TAB(12);"[DOWN]";AXIS$
1810 FOR Y=15 TO 1 STEP -1
1820 PRINT TAB(9);"[c 3]";RIGHT$(" "+STR
$(Y),2);" ";
1830 FOR X=1 TO 15
1840 PP%=PEEK(BOARD+16*Y+X)
1850 IF PP%=1 THEN PRINT"[c 3][s Q]";:GO
TO 1880
1860 IF PP%=2 THEN PRINT"[WHITE][s Q]";:
GOTO 1880
1870 PRINT"[c 6][s +]";
1880 NEXT
1890 PRINT"[c 3] ";RIGHT$(" "+STR$(Y),2)
1900 NEXT
1910 PRINT TAB(12);AXIS$
1920 PRINT"[DOWN][YELLOW]     LAST MOVE:";
1930 ITCP%=LOCAT%:GOSUB 2260:PRINT TAB(2
4);A1$
1940 PRINT TAB(18);"[DOWN][CYAN]";A2$
1950 RETURN
1990 REM INPUT ROUTINE
2000 IS%=0
2010 POKE 780,0:POKE 781,IP%+1:POKE 782,
39:SYS 65520
2015 FOR II=1 TO 79:PRINT CHR$(20);:NEXT
2020 A$=""
2030 PRINT"[WHITE]    ";MESS$(IM%);" ";
2060 GET I$:IF I$="" GOTO 2060
2065 IF ASC(I$)=13 GOTO 2120
2070 IF ASC(I$)<>20 GOTO 2100
2080 IF IS%>0 THEN IS%=IS%-1:A$=LEFT$(A$
,IS%):PRINT I$;:GOTO 2060
2085 GOTO 2060
2100 IF IS%<IW% THEN IS%=IS%+1:A$=A$+I$:
PRINT"[CYAN]";I$;
2110 GOTO 2060
2120 RETURN
2160 REM MESSAGE ROUTINE
2190 POKE 780,0:POKE 781,MP%:POKE 782,39
:SYS 65520
2195 FOR ML=1 TO 39:PRINT CHR$(20);:NEXT
2200 PRINT"     [RVSON][c 3]";MESS$(MM%);
O$;"[RVSOFF]";
2220 RETURN
2260 REM INT-TO-CHAR ROUTINE
2270 IF ITCP%=0 THEN PRINT"[CYAN]HANDICA
P";:RETURN
2275 C$=CHR$(ITCP%-16*INT(ITCP%/16)+64)+
MID$(STR$(INT(ITCP%/16)),2)
2280 PRINT"[CYAN]";C$;"      ";:RETURN
```

Amstrad CPC 464/664:

```
10 MEMORY &9FFF
20 DIM s(300):REM stack
30 GOSUB 170:REM init
40 GOSUB 1270:REM introduction
50 CLS:GOSUB 1730:REM print board
60 mve%=mve%+1:REM white moves here
70 IF over% THEN 60
80 mve%=mve%+1:REM black moves here
90 IF NOT over% THEN 60
100 ip%=23:im%=9:iw%=1:GOSUB 1990
110 IF a$="Y" THEN 40
115 IF a$<>"N" THEN 100
120 mp%=25:mm%=5:o$="":GOSUB 216
0:REM message
130 END
170 REM init routine
190 black%=1:white%=2:colour%=3
195 INK 1,26:REM white
197 INK 2,24:REM yellow
200 marker%=4:liberty%=8
220 board=&A000
240 DIM capture%(2)
250 axis$="A B C D E F G H I J K
 L M N O"
260 GOSUB 390:REM read messages
290 DIM dir%(4)
300 RESTORE 340
310 FOR i%=1 TO 4
320 READ dir%(i%)
330 NEXT i%
340 DATA 16,1,-16,-1
350 RETURN
390 REM read message routine
410 RESTORE 460
420 DIM mess$(9)
430 FOR m%=0 TO 9
440 READ mess$(m%)
450 NEXT m%
460 DATA "O.K. THINKING..."
470 DATA "Illegal entry: "
480 DATA "Stone already on point
: "
490 DATA "Illegal. Ko on point: "
500 DATA "Illegal. Suicide on
point: "
510 DATA "    O.K. GAME OVER"
520 DATA ""
530 DATA "How many handicap ston
es may I          have (2-9) ?"
540 DATA "Type your move (eg. H8
) PASS or          QUIT :"
550 DATA "Do you want to play ag
ain (Y/N)?"
560 RETURN
1270 REM introduction routine
1280 GOSUB 1340:REM game init
1290 GOSUB 1450:REM title screen
1300 RETURN
1340 REM game init routine
1345 stack%=1
1360 atari1$="      ":atari2$="
1370 location%=0:mve%=1
1380 over%=0
1390 capture%(1)=0:capture%(2)=0
1410 RETURN
1450 REM title screen routine
1460 CLS
1462 FOR c=1 TO 20 STEP 3
1465 RESTORE 1474
1470 MOVE 260+c,320+c
1472 FOR i=1 TO 8:READ x,y:DRAWR
 x,y,3:NEXT i
1474 DATA -10,10,-50,0,-10,-10,0
,-80,10,-10,50,0,10,10,0,20
1476 MOVER -10,0:DRAWR 20,0
1478 MOVE 380+c,320+c
1480 FOR i=1 TO 8:READ x,y:DRAWR
 x,y:NEXT i
1482 DATA -10,10,-50,0,-10,-10,0
,-80,10,-10,50,0,10,10,0,80
1484 NEXT c
1500 LOCATE 12,13:PEN 3:PRINT
1510 LOCATE 1,16:PEN 2:PRINT "Yo
u will play the ";
1520 PEN 1:PRINT"white ";:PEN 2:
PRINT"stones,  and"
1530 PRINT"and the computer (bei
ng weaker) will"
1540 PRINT"have the ";:PEN 3:PRI
NT"red ";:PEN 2:PRINT"stones wit
```

```
h a  handicap"
1550 PRINT TAB(13)"advantage"
1560 ip%=22:im%=7:iw%=1:GOSUB 19
90:REM input
1565 hand%=ASC(a$)-48
1570 IF hand%<2 OR hand%>9 THEN
1560
1590 RETURN
1730 REM print board routine
1735 INK 0,4:BORDER 9:REM magent
a/green
1750 LOCATE 3,1:PEN 2:PRINT" Sto
nes captured by:";TAB(32)"Move"
1770 PRINT TAB(3)"White =";:PEN
3:PRINT capture%(2);
1780 PEN 2:PRINT"   Black =";:PE
N 3:PRINT capture%(1);
1790 PRINT TAB(33)mve%
1800 PRINT:PRINT TAB(5)axis$
1810 FOR y%=15 TO 1 STEP -1
1820 LOCATE 2,20-y%:PRINT RIGHT$
(" "+STR$(y%),2);
1830 FOR x%=1 TO 15
1835 PRINT" ";
1840 p%=PEEK(board+16*y%+x%)
1850 IF p%=1 THEN PEN 3:PRINT"O"
;:GOTO 1880
1860 IF p%=2 THEN PEN 1:PRINT"O"
;:GOTO 1880
1870 PEN 2:PRINT"+";
1880 NEXT x%
1890 PEN 3:PRINT TAB(34)y%
1900 NEXT y%
1910 PRINT TAB(5)axis$
1920 PEN 2:PRINT TAB(3)"My last
move was:";
1930 itcp%=location%:GOSUB 2260:
PEN 1:PRINT TAB(30) atari1$
1940 LOCATE 16,23:PRINT atari2$
1950 RETURN
1990 REM input routine
2000 is%=0
2010 LOCATE 3,ip%:PRINT SPACE$(62)
2020 a$=""
2030 LOCATE 3,ip%:PEN 3:PRINT me
ss$(im%);
2060 i$="":WHILE i$="":i$=INKEY$
:WEND
2065 IF i$=CHR$(13) THEN 2120
2070 IF i$<>CHR$(127) THEN 2090
2080 IF is%>0 THEN is%=is%-1:a$=
LEFT$(a$,is%):PRINT CHR$(8);" ";
CHR$(8);
2085 GOTO 2060
2090 IF i$>="a" AND i$<="z" THEN
 i$=CHR$(ASC(i$)-32)
2100 IF is%<iw% THEN is%=is%+1:a
$=a$+i$:PEN 2:PRINT i$;
2110 GOTO 2060
2120 RETURN
2160 REM message routine
2190 LOCATE 3,mp%:PRINT SPACE$(3
6);
2200 LOCATE 3,mp%:PEN 3:PRINT me
ss$(mm%);o$
2220 RETURN
2260 REM int-to-char routine
2270 IF itcp%=0 THEN PEN 1:PRINT
"Handicap";:RETURN
2275 c$=CHR$(itcp%-16*INT(itcp%/
16)+64):r$=MID$(STR$(INT(itcp%/1
6)),2)
2280 PEN 1:PRINT c$;r$;:RETURN
```

Sinclair Spectrum:

```
  10 CLEAR 63999
  20 DIM s(300)
  30 PRINT AT 10,10;"PLEASE WAIT
": GO SUB 170
  40 GO SUB 1270
  50 CLS : GO SUB 1730
  60 LET move=move+1: REM white
moves here
  70 IF end THEN  GO TO 100
  80 LET move=move+1: REM black
moves here
  90 IF NOT end THEN  GO TO 60
 100 LET ip=20: LET im=10: LET i
w=1: GO SUB 1990
 110 IF a$="Y" THEN  GO TO 40
 115 IF a$<>"N" THEN  GO TO 100
 120 LET mp=21: LET mm=6: LET o$
="": GO SUB 2160
 130 STOP
 170 REM initialise routine
 190 LET black=1: LET white=2: L
ET colour=3
 200 LET marker=4: LET liberty=8
 220 LET board=64000
 240 DIM c(2)
 260 GO SUB 390
 290 DIM d(4)
 300 RESTORE 340
 310 FOR l=1 TO 4
 320 READ d(l)
 330 NEXT l
 340 DATA 16,1,-16,-1
 350 RETURN
 390 REM read-messages routine
 410 RESTORE 460
 420 DIM m$(10,43)
 430 FOR m=1 TO 10
 440 READ m$(m)
 450 NEXT m
 460 DATA "O.K. THINKING..."
 470 DATA "Illegal entry: "
 480 DATA "Stone already on poin
t: "
 490 DATA "Illegal. Ko on point:"
 500 DATA "Illegal. Suicide on p
oint: "
 510 DATA "    O.K. GAME OVER."
 520 DATA ""
 530 DATA "How many handicap sto
nes may I  have (2-9) ?"
 540 DATA "Type your move (eg. H
8), PASS   or QUIT :"
 550 DATA "Do you want to play a
gain       (Y/N) ?"
 560 RETURN
1260:
1270 REM introduction routine
1280 GO SUB 1340
1290 GO SUB 1450
1300 RETURN
1340 REM game-init routine
1345 LET stack=1
1360 LET x$="     ": LET y$="
 "
1370 LET location=0: LET move=1
1380 LET end=0
1390 LET c(1)=0: LET c(2)=0
1410 RETURN
1450 REM title-screen routine
1460 BORDER 0: PAPER 0: CLS
1470 INK 2: PRINT AT 3,12;"sh23s
h1  sh23sh1"
1480 PRINT AT 4,12;"sh84sh3  sh8
 sh8"
1490 PRINT AT 5,12;"sh8  sh8  sh8
 sh8"
1500 PRINT AT 6,12;"132  132"
1510 PRINT AT 8,7; INK 5;"by  Ma
rcus Jeffery"
1520 PRINT : PRINT  INK 3;"You w
ill play the "; INK 7;"white"; I
NK 3;" stones,"
1530 PRINT  INK 3;"and the compu
ter (being weaker)"
1540 PRINT  INK 3;"will have the
 "; INK 2;" red  "; INK 3;"stone
s with"
1550 PRINT  INK 3;"     a handic
ap advantage."
1560 LET ip=15: LET im=8: LET iw
=1: GO SUB 1990
1565 LET hand=CODE (a$)-48
1570 IF hand<2 OR hand>9 THEN  G
O TO 1560
1590 RETURN
1730 REM print-board routine
1750 PRINT AT 0,0; INK 6;" Stone
s captured by:";
1760 PRINT  INK 7;"      Move "
1770 PRINT  INK 6;" White="; INK
 5;c(2);
1780 PRINT TAB 10; INK 6;"  Blac
k ="; INK 5;c(1);
1790 PRINT TAB 27; INK 7;move
1810 FOR y=15 TO 1 STEP -1
1820 PRINT AT 17-y,5; INK 2;(" "
+STR$ y)(LEN STR$ y TO );" ";
1830 FOR x=1 TO 15
1840 LET pp=PEEK (board+16*y+x)
1850 IF pp=1 THEN  PRINT  PAPER
4; INK 2;"O";: GO TO 1880
1860 IF pp=2 THEN  PRINT  PAPER
4; INK 7;"O";: GO TO 1880
1870 PRINT  PAPER 4; INK 0;"+";
1880 NEXT x
1890 PRINT  INK 2;" ";y
1900 NEXT y
1910 PRINT TAB 8; INK 2;"ABCDEFG
HIJKLMNO"
1920 PRINT  INK 6;"Last move: ";
1930 LET itcp=location: GO SUB 2
260: PRINT TAB 20; INK 5;x$
1940 PRINT AT 20,20; INK 5;y$: B
EEP 1,0
1950 RETURN
1990 REM input routine
2000 LET is=0
2010 PRINT AT ip,0;: FOR i=1 TO
62: PRINT " ";: NEXT i
2020 LET a$=""
2030 PRINT AT ip,0; INK 7;m$(im)
;
2060 LET i$=INKEY$: IF i$="" THE
N  GO TO 2060
2065 IF CODE i$=13 THEN  GO TO 2
120
2070 IF CODE i$<>12 THEN  GO TO
2090
2080 IF is>0 THEN  LET is=is-1:
LET a$=a$(1 TO LEN a$-1): PRINT
CHR$ 8;" ";CHR$ 8;: GO TO 2060
2085 GO TO 2060
2090 IF i$>="a" AND i$<="z" THEN
  LET i$=CHR$ (CODE i$-32)
2100 IF is<iw THEN  LET is=is+1:
 LET a$=a$+i$: PRINT  INK 5;i$;
2110 GO TO 2060
2120 RETURN
2160 REM message routine
2190 PRINT AT mp,0;: FOR m=1 TO
30: PRINT " ";: NEXT m
2200 PRINT AT mp,0; INK 2;m$(mm,
 TO 26);o$;
2220 RETURN
2260 REM int-to-char routine
2270 IF itcp=0 THEN  PRINT  INK
5;"Handicap";: RETURN
2275 LET c$=CHR$ (itcp-16*INT (i
tcp/16)+64): LET r$=STR$ (INT (i
tcp/16))
2280 PRINT  INK 5;c$;r$;: RETURN
```

QUERY LANGUAGE

A *query language* is a component of the operating system of a database management (DBM) system. It directly accesses, or 'queries', the database itself. When asked to obtain some information from the database, for example, the query language will interrogate it in order to find a match within the parameters that have been set. Often, the interrogation may be performed directly from the keyboard — such a system is known as an 'interactive query language'.

It should be noted that a query language is basically a list of commands that will be interpreted and acted upon by the computer. While the query language commands will produce a formula by which the computer will interrogate the database, the user should not be concerned with exactly how the system of retrieval is to take place. In other words, the information retrieval should be 'transparent' to the user. Some of the more sophisticated query languages will not only search the database for a result that will be displayed on the VDU screen, but also, by using conditional statements, allow for further searches to be initiated.

Query languages vary in quality and complexity depending on the nature of the DBM itself. Some merely have simple search routines, whereas others may have complicated built-in cross referencing facilities.

QUEUE

A queue is a First In First Out (FIFO, see page 576) data structure, as opposed to a 'stack' system in which data is organised on a First In Last Out (FILO) basis.

Queues are most often used in buffer systems, particularly with regard to printers. In this case, the information arrives from the computer far more rapidly than can be processed by the printer. A queue system is therefore implemented to store the information as it waits to be sent through. Obviously, the information will have to be stored in strict sequence, otherwise the words will appear as a meaningless jumble on hard copy.

QUICKSORT

A *quicksort* is a method of sorting that, in a list of items to be sorted, compares the top and bottom items with the items above and below them, respectively. Items are exchanged in the list, according to the user's dictates, as the program proceeds up and down the file. This process is repeated until the sort is complete.

RADIX

Each number system has a number of distinct digits; the decimal system, for example, has 10 while the hexadecimal system has 16. The *radix* (also known as the 'base' — see page 148) is the number of different digits, including zero, used in a particular number system. The use of the radix has many applications in the mathematics of computing and has generated a host of related terms.

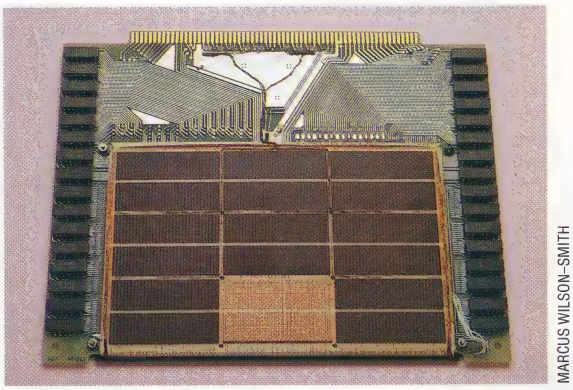
The *radix point* (also known as 'radix notation') is a character or symbol, usually a full stop, which separates the integer in a real number from its traditional part. Examples of this are 12.5, 1000101.01 and 23.FF.

Among other applications is 'radix sorting'. In a list of numbers, radix sorting involves finding the least significant digits of each of the numbers in the list, sorting them and then moving onto the next least significant digit until the most significant number is reached.

An alternative method of sorting is the 'radix exchange'. Here, the most significant digits in each number are compared, and then sorted into subfiles, which are broken down further by examination of the next most significant digits. This process continues until the file is sorted.

RAM

Also known as *random-access memory*, this is one of the most important parts of a computer system. A *RAM* chip is a semiconductor device made up of a number of individual cells, often in two-dimensional matrices, each of which is able to hold a one or a zero. Each cell can be addressed either singly or, more commonly, in groups or 'words' by the processor. The processor can read the value in the cell or else write a new value to it.



MARCUS WILSON-SMITH

Memories Of Old
Advances in chip technology have dramatically reduced the cost per byte of random access memory. The board shown here, from a Computer Automation Inc. minicomputer built in 1976, uses magnetic core technology to store around 2K of memory on a board that could nowadays house 64 256K RAM chips. Note the driver chips around the edge of the board; each sends current along the copper wires to reverse the magnetic polarity of the relevant bit

There are two main types of RAM chip. *dynamic RAM* holds the values in each of the cells by means of electrical capacitance, which needs to be regularly recharged or 'refreshed'. The alternative is *static RAM*, which holds one or zero in a cell depending on whether the 'latch' ('flip-flop', see page 608) on the cell is open or closed. Dynamic RAM has the advantage of greater packing density, whereas static RAM has faster access.

In the early 1970s, 16 Kbytes of RAM (131,072 individual cells) was considered a large memory. Today, a standard business machine will be sold fitted with a minimum of 128 Kbytes, and many will have the capacity to contain up to one megabyte of data.

MOUSE PIECE

Since the introduction of the Apple Macintosh, with its mouse-based operating system, the micro industry has cast aside its initial doubts and eagerly incorporated this technology into the home micro market. We look here at the SMC Supplies Magic Mouse, the first such device for the Commodore 64.

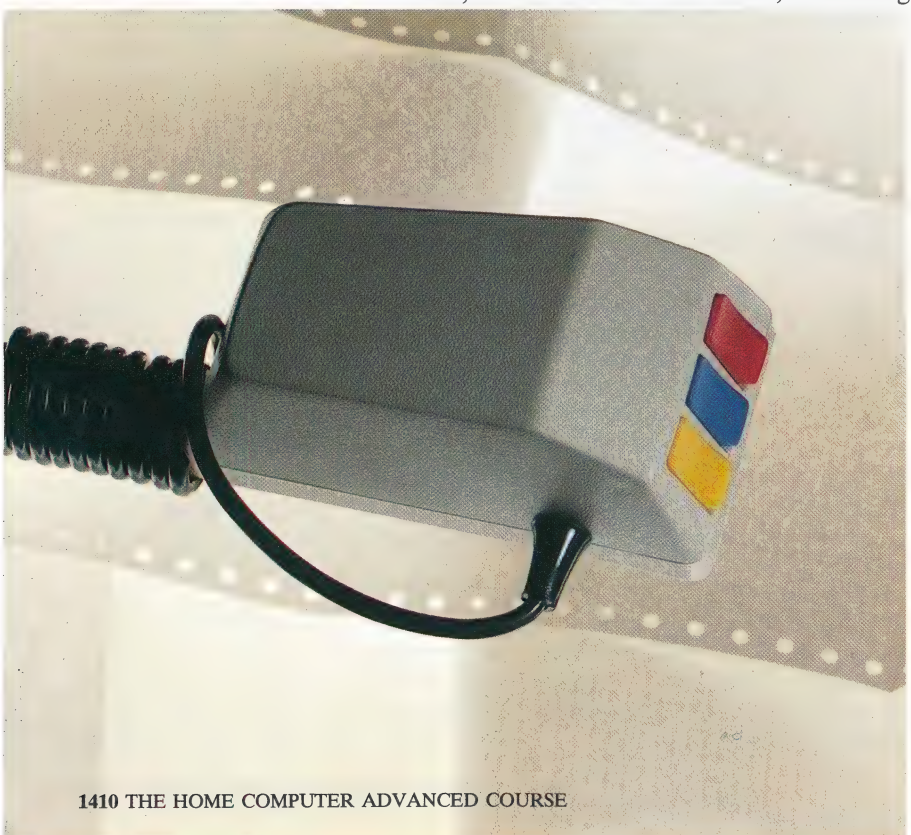
Considering the current popularity of mouse-based operating systems, it is perhaps hard to believe that initially there was much debate within the microcomputer industry as to whether a mouse-based system was acceptable, let alone desirable. The comparative failure of the Apple Lisa, the first micro to have a mouse-based operating system, seemed to confirm the view that end users did not want these systems.

The later success of the Macintosh silenced the critics, however, and justified Apple's faith in its system. Attitudes have turned around drastically; computer manufacturers are racing to produce mouse-based machines, and many third party manufacturers are issuing mouse packages for existing machines.

The Magic Mouse from SMC Supplies is one of the first such products to become available for the Commodore 64. Included with the mouse is software, on both cassette and disk, containing four application packages. The SMC mouse is larger than most similar devices and at 125 by 66 by 50mm (about 5 by 2¾ by 2in) it is almost twice the size of the Apple mouse. The Magic Mouse also has three coloured buttons on the front as opposed to the more often used two. The buttons are used to select functions which, of course, vary depending on the applications being used. Inside is a hard rubber ball pressing against a pair of shaft encoders.

Apparently, the original design called for a standard steel ball bearing but this was found to be unsuitable, and the launch of the device was delayed to allow the rubber replacements to be fitted. However, the mouse appears to work perfectly.

It's Magic
The SMC Magic Mouse is one of the first mouse-based systems to be developed for the Commodore 64. Although it is not designed as a true WIMP (Windows, Icons, Mice, Programming) package, it can be usefully employed as a development tool for programmers. The mouse plugs into the Commodore 64's joystick port and the icons on screen are selected by using the three buttons fitted on the mouse



CRISPIN THOMAS

UTILITY PROGRAMS

The Magic Mouse package is not an operating system like that of the Apple Macintosh, but rather a graphics design facility. The four programs provided with the mouse are Hi-Res Designer, Sprite Designer, Icon Designer and Mouse Controller.

In order to be able to run the software properly, the system file must first be LOADed and the mouse cursor calibrated. The disk-based system is LOADed automatically with a main menu that enables you to select any of the four utility programs. The cassette-based system is slightly different. Here, the system file is part of the Hi-Res Designer and so this program must be LOADed before you can use any of the other three. Also, the cursor must be positioned on the bottom right-hand corner of the screen before starting out.

The Hi-Res Designer is a 'paintbox' program looking very similar to that of the Koala Pad software (see page 629). When it is loaded, the screen changes to a series of boxes, each of which contains a different option, such as Box, Draw or Fill. By moving the mouse cursor to one of the boxes and pressing the red button on the mouse, this option will be chosen. Similarly, in order to choose either the current foreground or background colour, you simply move the cursor to the lines of 16 available colours and select one. To move to the 'canvas', you press the blue button.

The Icon and Sprite Designer packages are very similar and only really differ in their end applications. Like most programs of this nature, the Sprite Designer presents you with a 24-column by 21-row grid. Pixels are turned on by pressing the red button when the cursor is over the appropriate position. The sprite itself appears in a window in the top right-hand corner of the screen, and when the cursor moves off the grid, a number

of options appear in the bottom right-hand corner. These are for such applications as expanding the sprite, changing its colour or moving on to a different sprite.

The Icon Designer appears almost identical, with a grid in the corner and the icon appearing in a window at the top. The difference between them is that the Icon Designer is used to construct user-defined graphics, such as founts and so on, which can be SAVEd and reused.

The final program provided by the Magic Mouse software is the Mouse Controller. This program does not *do* anything itself, but rather provides the driver software that enables your programs to be run from the system. Although at first sight this is the least useful of the four programs, it does, in fact, have the most potential, since it will encourage you to write your own software for the mouse.

The great advantage that mouse-based systems appear to have over digitisers is the steadiness of the cursor, which allows accurate positioning of the lines and colours you wish to draw. A digitiser is based around a grid of interconnecting nodes; however, its resolution is less than the screen's. Thus when the 'pen' is between two nodes, the computer cannot decide which position is intended and so it tends to jump between the two. This, of course, can often have disastrous results for the picture you are attempting to construct. When using the mouse, however, its cursor reacts only to changes in its potentiometers caused by movements in the ball, and so the cursor will remain steady if the mouse is not moved.

The system is not perfect, however. Some aspects of the Hi-Res Designer are not as friendly as those in the other packages. For example, the RUB command only allows you to delete errors pixel by pixel. This is time-consuming and liable to generate further errors if you delete the wrong pixels. A much better method is one that deletes all additions to the canvas since you last went to the menu screen. This means, of course, that a lot of good work will be deleted in the process, but at least you will be able to start with a clean sheet.

The Magic Mouse manual provides an excellent tutorial. Each of the programs has both a full explanation of how the development software works, and programming information to allow you to incorporate the finished product into your own software — including sample programs. This shows a great deal of foresight on the part of SMC, and it should be congratulated for its thorough and useful manual.

Although the SMC Magic Mouse is far from being the Macintosh Emulator, the company has obviously developed the initial software as a programmer's tool, similar to the AMX mouse (see page 1050) or the DR GEM software. The company doubtless hopes that programmers will produce additional software that will consequently enhance the system's attraction. This approach may not produce quick sales, but it is likely to ensure a long life for the Magic Mouse.

Artistic Development
These pictures were built up using the Hi-Res Designer application program for use with the SMC Magic Mouse. The program allows the usual commands that are available with these types of packages, including LINE, DRAW, FILL and CIRCLE. Up to 16 different colours can be used to make up the picture, and a wide range of different 'brushes' are available, which vary the thicknesses and textures of the lines

MAGIC MOUSE

PRICE
£59.95 inc VAT

INTERFACE
The Magic Mouse plugs into the Commodore 64's joystick port

SOFTWARE
The mouse is supplied with four applications programs on both disk and cassette

DOCUMENTATION
The manual gives full details on how to set up the mouse, use the existing software and incorporate the results into your own programs

STRENGTHS
The mouse appears well-built and reliable and users should have little trouble transferring the sprites and user-defined graphics into their own programs

WEAKNESSES
The total package lacks the scope of many other mouse-orientated programs and may find itself left behind by more sophisticated packages

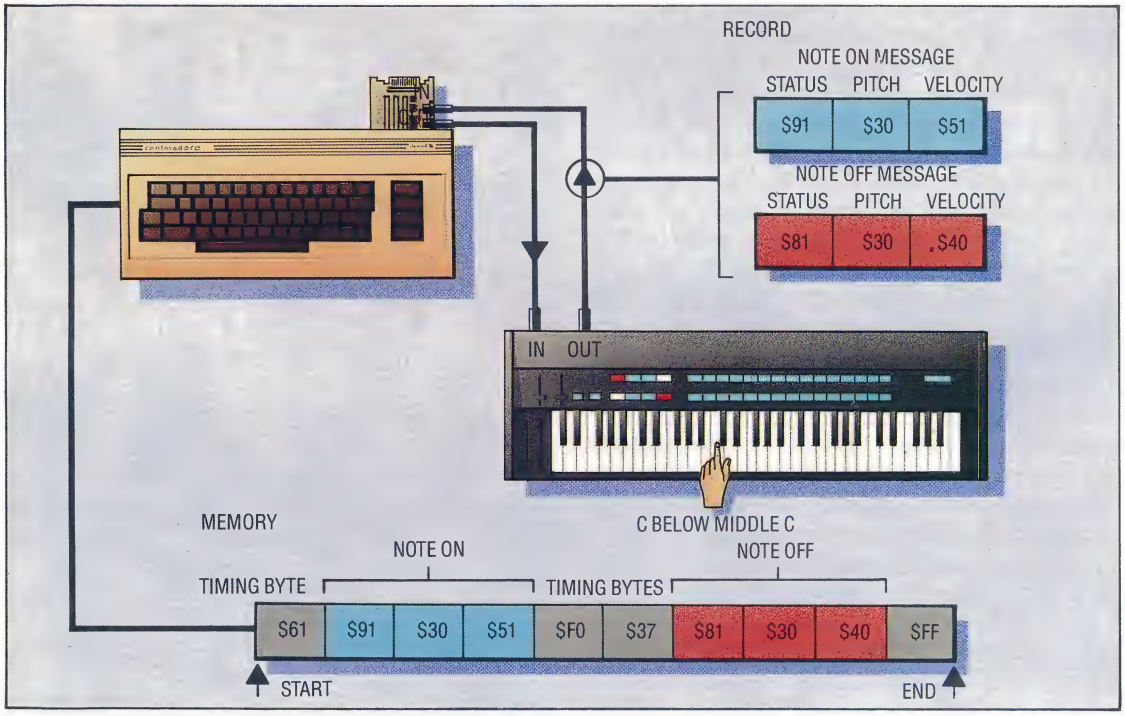
JOHN CLEMENTSON

SCREEN DUMPS BY DIMENSION GRAPHICS

HIGH FIDELITY

Having discussed the hardware and software specification for our MIDI interface and constructed the board, we turn our attention to the development of a program that will allow the computer to act as a real-time digital recorder and playback device.

which a computer can be used to interact with MIDI instruments by tackling the relatively straightforward task of designing a piece of software that can record a tune played in real-time in the computer's memory. Then, in response to an instruction, the program will replay the tune over MIDI back to the synthesiser. This is, of course, a



His Micro's Voice
Our diagram shows how a single note would be recorded using the Digital Recorder program. During the recording phase the synthesiser sends a 'note on' message as the key is pressed and a 'note off' message when the key is released. These messages are stored in memory, together with timing information that records how many 2 millisecond periods elapse between one event and the next. The timing information between the note on and note off messages is held in two bytes in our example, showing how timer overflow is dealt with. During playback the program uses the timing information to delay transmission of the note messages so that the same 'articulation' is reproduced

When the keys on a MIDI keyboard are pressed, or a controlling device such as a pitch bending wheel is activated, then these 'events' are transmitted in the form of data bytes to the MIDI OUT socket on the back of the synthesiser. Much of the previous instalment (see page 1386) was concerned with giving details about the meanings of these byte values so that we can write programs to accept MIDI data and process it. The computer can also be used as a source of MIDI data, which can be transmitted to a synthesiser in order to activate its sound generating circuits. An obvious example of this second use is to set the computer up as a sequencer, allowing musical sequences to be entered and edited in 'step-time' (programming individual steps within a bar) from the computer keyboard. This is done prior to transmission via MIDI to activate the synthesiser that will play the tune. In fact, this is a relatively simple task; MIDI is capable of much more. Imagine a sequencer that used all 16 MIDI channels to control an array of different keyboards and a drum machine!

We can combine several aspects of the way in digital recording.

Although superficially similar to analogue recording using a tape recorder and magnetic tape, digital recording via MIDI is radically different. When a recording is made on a piece of tape the actual musical sounds are encoded into magnetic patterns on the tape surface, and the position of these patterns on the tape determines the order in which the sounds are to be reproduced during playback, as well as the time intervals between one sound and the next. Imagine two notes being recorded in this way where there was, say, a two-second gap between the first note and the second. In this interval, the tape would continue to pass the tape heads although nothing would be recorded. When the tape was replayed, the length of this blank section of tape would determine the time interval between the listener hearing the first note and the second.

In a digital recording system that uses MIDI, the two notes would correspond to two keypress events, each generating a MIDI message. The problem with a real-time recording is that we must

have some way of recording the time interval between the computer receiving the first MIDI message and the second. Using the tape recording analogy the obvious way to do this would be to store the messages in a memory array, storing 'blank' byte values for each time unit in the interval. Thus, during playback, the two notes could be played back with the correct intervening time interval. It is not difficult to see that this method of recording MIDI events in real-time is potentially very wasteful of precious memory.

An alternative to this simplistic analogy of an analogue tape recorder is to record timing information in byte form rather than 'blank' byte values. For example, if there are 50 time units between the first note and the second then a timing byte containing the value 50 will be entered in the memory array rather than 50 'blank' bytes.

This achieves a considerable memory saving, but also makes it difficult to predict the maximum length of a tune that can be recorded in a given amount of memory. Tunes that involve fast fingerwork, for example, will require more bytes to record the large number of note on/off messages transmitted by the synthesiser than a slower piece of the same length. Using the pitch-wheel also tends to eat up large amounts of memory because of the number of MIDI messages its operation generates.

DIGITAL RECORDER PROGRAM

In the program we provide here, we will therefore need some kind of counter to record the time intervals between events. The counter we use is a variable called CLOCKS. The rate at which this counter is incremented sets the accuracy with which the timing can be determined, and thus the fidelity of the recording. A trade-off results when selecting the optimal time interval — we have chosen an interval of about two milliseconds. This is not so small as to be less then the execution times of the routines, but not so large as to adversely affect the accuracy of the recording. This time interval is called the *resolution* of the recorder, since it is the smallest time interval that can be recorded. The interval is generated by a timer on one of the CIA or VIA chips on the micro.

The MIDI message format must be slightly extended to take timing into account. This is done by placing before each MIDI message a single byte that represents the number of time intervals that have passed since the last message. This single byte can record up to 239 time units, and so can have values from $0 to $EF.

If 240 intervals pass without a message being received, an overflow message is recorded. This overflow message is the single byte $F0. Finally, a single byte message, $FF, has been chosen to mark the end of the sequence. Byte values $F1 to $FE are reserved for your own program markers.

In the final section of this MIDI project we will give a version of the Digital Recorder program for the BBC Micro and point the way for aspiring MIDI programmers to implement their ideas.

PHOTO OF PIANOLLA ROLLS : MARCUS WILSON-SMITH

Digital Recorder Program

BASIC Loader

```
10 FOR I = 49152 TO 49581
20 READ X:POKE I,X:S=S+X:NEXT
30 READ X:IF S=X THEN PRINT"OK SYS49152 TO RUN":END
40 PRINT"CHECKSUM ERROR"
50 END
100 DATA76,7,192,2,83,201,241,169,3
110 DATA141,0,222,169,21,141,0,222,169
120 DATA255,141,173,193,169,0,141,6
130 DATA220,169,8,141,7,220,169,17,141
140 DATA15,220,32,228,255,201,0,240
150 DATA249,201,69,208,1,96,201,82,240
160 DATA4,201,80,208,236,72,120,169
170 DATA127,141,0,220,169,173,133,247
180 DATA169,193,133,248,169,173,141,5
190 DATA192,169,241,141,6,192,160,0
200 DATA104,201,82,8,240,66,169,2,32
210 DATA69,193,32,44,193,201,255,208,4
220 DATA40,76,4,193,141,4,192,201,240
230 DATA173,4,192,240,12,32,246,192
240 DATA240,251,206,4,192,208,246,176
250 DATA223,32,44,193,72,173,0,222,41
260 DATA2,240,249,104,141,1,222,16,3
270 DATA32,53,193,202,208,233,174,3
280 DATA192,16,195,169,1,32,69,193,140
290 DATA4,192,162,255,32,246,192,240
300 DATA16,238,4,192,173,4,192,201,240
310 DATA144,6,32,21,193,140,4,192,173
320 DATA0,222,41,1,208,6,224,1,48,224
330 DATA16,243,173,1,222,48,11,224,0
340 DATA48,213,208,27,174,3,192,16,11
350 DATA201,248,176,223,201,240,176
360 DATA196,32,53,193,72,173,4,192,32
370 DATA21,193,140,4,192,104,32,21,193
380 DATA202,240,178,208,197,173,1,220
390 DATA73,239,208,18,104,104,40,208,4
400 DATA169,255,145,247,88,169,0,32,69
410 DATA193,76,37,192,173,13,220,41,2
420 DATA96,145,247,238,5,192,208,18
430 DATA238,6,192,208,13,104,104,104
440 DATA169,3,32,69,193,76,6,193,177
450 DATA247,290,247,208,2,230,248,96
460 DATA162,2,201,192,144,5,201,224
470 DATA176,1,202,142,3,192,232,96,10
480 DATA170,189,96,193,133,249,189,97
490 DATA193,133,250,160,0,177,249,240
500 DATA6,32,210,255,200,208,246,160,0
510 DATA96,104,193,134,193,146,193,158
520 DATA193,82,32,61,32,82,69,67,79,82
530 DATA68,44,80,32,61,32,80,76,65,89
540 DATA44,69,32,61,32,69,88,73,84,13
550 DATA0,147,82,69,67,79,82,68,73,78
560 DATA71,13,0,147,80,76,65,89,66,65
570 DATA67,75,32,13,0,78,85,84,32,79
580 DATA70,32,77,69,77,79,82,89,13,0
590 DATA240
600 DATA52178:REM*CHECKSUM*
```

Assembly Listing

```
*=$C000
GETIN  = $FFE4          ;INPUT A CHARACTER
       CHROUT   = $FFD2 ;OUTPUT A CHARACTER
       JMP START
       NOBYTS *=*+1     ;NO. BYTES IN CURRENT MESSAGE
       CLOCKS  *=*+1    ;NO 2MS PERIODS SINCE LAST MESSA

       FREMEM *=*+2     ;- NO. OF FREE BYTES
       DATREG  = $DE01  ;TRANSMIT/RECEIVE REG ON ACIA
       STREG   = $DE00  ;STATUS/CONTROL REG FOR ACIA
       STOPKY = $EF
       MEM     = $F7    ;DATA MEMORY POINTER
       PTR     = $F9    ;STRING READ POINTER
       START   = *
;++++ SET UP ROUTINES ++++
       LDA #$03
       STA STREG        ;INITIALISE 6850
       LDA #$16
       STA STREG
       G10     LDA #$FF
```

Program Operation

Operation of the Digital Recorder program for the Commodore 64 is straightforward. Selecting the record option will record events arriving at MIDI IN. Recording stops when either the available memory is used up or the Space bar is pressed. The playback option will then reproduce the recorded sequence of events at MIDI OUT. Playback continues until either the Space bar is pressed or the end of the sequence is reached. A subsequent recording will overwrite the old one

```
STA     LOWMEM          ;INIT TO EMPTY
        LDA #$00
        STA $DC06
        LDA #$08
        STA $DC07
        LDA #$11
        STA $DC0F       ;START TIMER B
G20         JSR GETIN
        CMP #$00
        BEQ G20
        CMP #69         ;PRESS 'E' TO EXIT
        BNE G22
        RTS
G22     = *             ;NOT EXIT
        CMP #82
        BEQ G30
        CMP #80
        BNE G20
                        ;++++ REAL TIME PROG STARTS HERE
G30         PHA         ;SAVE RECORD/PB COMMAND
        SEI
        LDA #$7F
        STA $DC00       ;SCAN LAST ROW OF KEYS
        LDA #<LOWMEM
        STA MEM
        LDA #>LOWMEM
        STA MEM+1
        LDA #<FRBYTS
        STA FREMEM
        LDA #>FRBYTS
STA     FREMEM+1
        LDY #$00
        PLA
        CMP #82
        PHP             ;SAVE Z FLAG = RECORD
        BEQ R00
                        ;
        P00 = *         ;PLAY ROUTINE
                        ;
        LDA #$02
        JSR STROUT
        P05 JSR READ
        CMP #$FF
        BNE P07         ;EXIT IF END REACHED
        PLP             ;ADJUST STACK
        JMP C20         ;EXIT PROCESS
        P07 STA CLOCKS
        CMP #$F0        ;CARRY SET = $F0 READ
        LDA CLOCKS
        BEQ P30         ;JMP IF NO TIME TO WAIT
        P10 JSR CHECK
        BEQ P10
        DEC CLOCKS
        BNE P10         ;JMP IF NOT TIMED OUT
        BCS P05         ;READ NEXT CLOCK IF $F0
                        ;
        P30 = *         ;READ NEXT MESSAGE BYTE
JSR     READ
PHA                     ;SAVE DATA BYTE
;
P35     = *             ;WAIT FOR TX REGISTER
LDA     STREG
AND     #$02
BEQ     P35             ;JMP IF STILL FULL
;
PLA                     ;GET DATA
STA     DATREG          ;SEND TO MIDI
BPL     P50             ;JMP IF NOT STATUS BYTE
JSR     GETNO           ;GET NO OF DATA BYTES
P50     DEX             ;DEC. BYTE COUNT
BNE     P30             ;GET NEXT DATA BYTE
LDX     NOBYTS
BPL     P05             ;GET CLOCK BYTE
;
R00     = *             ;RECORD ROUTINE
;
LDA     #$01
JSR     STROUT
STY     CLOCKS          ;CLOCKS = 0
R05     LDX #$FF        ;INIT TO NO DATA
R10     JSR CHECK
BEQ     R20
INC     CLOCKS
LDA     CLOCKS
CMP     #$F0
BCC     R20
JSR     STORE
STY     CLOCKS
;
R20     = *             ;CHECK FOR MIDI DATA
LDA     STREG
AND     #$01
BNE     R40             ;JMP IF DATA
CPX     #$01
BMI     R10             ;JMP IF WAITING
BPL     R20             ;JMP IF MID-MESSAGE
;
R40     = *             ;GET THE DATA
LDA     DATREG
BMI     R50             ;JMP IF STATUS BYTE
CPX     #$00
BMI     R10             ;JMP IF NOT CHANNEL DATA
BNE     R80             ;JMP IF MID-MESSAGE
LDX     NOBYTS          ;RESET BYTE COUNT
BPL     R60
;
R50     = *             ;STATUS BYTE PROCESS
CMP     #$F8
BCS     R20             ;IGNORE IF SYS REALTIME
CMP     #$F0
BCS     R05             ;JMP IF NOT CHANNEL STAT.
JSR     GETNO           ;GET NO OF DATA BYTES
;
```

```
R60     = *             ;RECORD THE DATA
PHA
LDA     CLOCKS
JSR     STORE
STY     CLOCKS          ;RESET COUNTER
PLA
;
R80     = *             ;RECORD MIDI DATA
JSR     STORE
DEX
BEQ     R10
BNE     R20
;
CHECK   = *
;
LDA     $DC01
EOR     #STOPKY         ;DOESN'T AFFECT CARRY
BNE     C40
PLA
PLA                     ;ADJUST STACK
PLP                     ;CHECK FOR REC/PB
BNE     C20             ;JMP IF PLAYBACK
LDA     #$FF
STA     (MEM),Y
C20     = *
CLI
LDA     #$00
JSR     STROUT
JMP     G20             ;WAIT FOR NEXT COMMAND
;
C40     = *             ;CHECK FOR TIMER B
LDA     $DC0D
AND     #$02            ;Z FLAG = NOT TIMED OUT
RTS
;
STORE   = *             ;STORE DATA IN MEMORY
;
STA     (MEM),Y
INC     FREMEM          ;CHECK MEMORY FREE
BNE     POINT
INC     FREMEM+1
BNE     POINT
PLA
PLA
PLA
LDA     #$03            ;ERROR MESSAGE
JSR     STROUT
JMP     C20             ;RESET PROGRAM
;
READ    = *             ;READ DATA FROM MEMORY
;
LDA     (MEM),Y
;
POINT   = *             ;UPDATE MEMORY POINTER
;
INC     MEM
BNE     P20
INC     MEM+1
P20     RTS
;
GETNO   = *             ;GET NO OF DATA BYTES
;           FOR STATUS IN ACC.
;
LDX     #$02            ;INIT TO 2 DATA BYTES
CMP     #$C0
BCC     N10
CMP     #$E0
BCS     N10
DEX                     ;1 DATA BYTE IF C0<=A<E0
N10     STX NOBYTS
INX                     ;ADD 1 FOR STATUS
RTS
;
STROUT  = *             ;PRINT STRING
;
ASL     A
TAX
LDA     MESTAB,X
STA     PTR
LDA     MESTAB+1,X
STA     PTR+1
LDY     #$00
M10     LDA (PTR),Y
BEQ     M70
JSR     CHROUT
INY
BNE     M10
M70     LDY #$00
RTS
;
;
MESTAB  = *             ;MESSAGE TABLE
;
.WORD   MESS0
.WORD   MESS1
.WORD   MESS2
.WORD   MESS3
;
MESS0   .BYTE 'R = RECORD,P = PLAY,E = EXIT',$0D,0

MESS1   .BYTE 147,'RECORDING',$0D,0


MESS2   .BYTE 147,'PLAYBACK',$0D,0


MESS3   .BYTE 'OUT OF MEMORY',$0D,0

LOWMEM  = *             ;START OF DATA MEMORY
FRBYTS  = LOWMEM - $D000
```

ACTIVE LIST

In this last instalment of our series on LISP, we look at a few more functions normally found in the language and demonstrate them with one or two slightly more complex examples. First, however, let's see why it is often useful to know how LISP represents data structures.

In order to have a uniform notation that can easily deal with identifiers, characters, numbers and lists in the same manner, LISP refers to everything via 'pointers'. Pointers can be easily simulated in any language, including BASIC. For instance, if you give the instruction:

```
POKE 100,55 : POKE 99,100 (Most micros)
```

or:

```
?100 = 55 : ?99 = 100 (BBC Micro/Electron)
```

then locations 100 and 99 would contain the given values. It is possible to consider the value in location 99 as a pointer to location 100. So if we were to give the command:

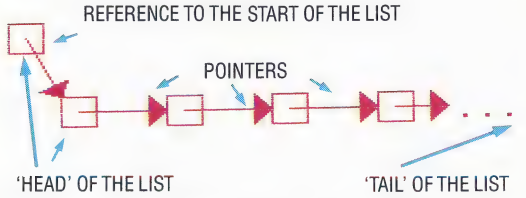
```
PRINT PEEK (PEEK(99))
```

or:

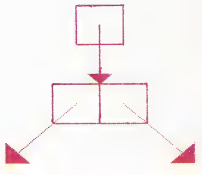
```
PRINT ?(?99)
```

we would get the answer 55. Note that on some micros, locations 99 and 100 may be in ROM, so will not change.

It is, of course, possible to create pointers to anywhere in memory, but if the location lies above 255, then it will be necessary to combine two bytes to give the range 0 to 65535. Now, if we assume that, in turn, the value 55 is a pointer to a further location, and so on, we will get the list structure as shown.

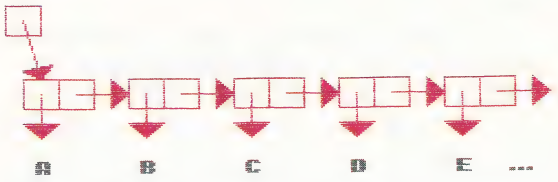


Unfortunately, this is of little use as yet because the list holds no information. This is achieved by combining locations to give two pointers at each node.



We can now use this to represent the list:

```
(A B C D E . . .)
```



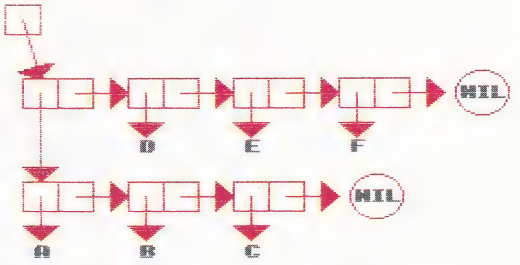
This is fine for infinite lists, but we must be able to stop them. This is easily done by having the final pointer refer to the empty list (or NIL construct), which we have seen previously (see page 1384). Consequently, we get the finite list:

```
(A B C D)
```

A list containing a list simply means that one of the left-hand pointers points to a new list, instead of an atom as in the previous case. So the list:

```
((A B C) D E F)
```

would be represented as shown:



It should now be clear how the functions CAR and CDR work. CAR (the head of its argument) is just the left-hand pointer of the first node of its argument. Conversely, CDR (everything but the head of its argument) is merely the right-hand pointer of the first node. So, if we call the complete list L, then:

```
CONS((CAR L) (CDR L))
```

gives the original list L, as its result.

We can thus see that CONS is a method of building lists. It forms its two arguments into a new list and returns as its result a pointer to the first item of the new list. It is possible to create any list structure using CONS, though the number of CONS calls can become tedious — hence the shorthand function LIST.

It should be noted that it is always preferable to

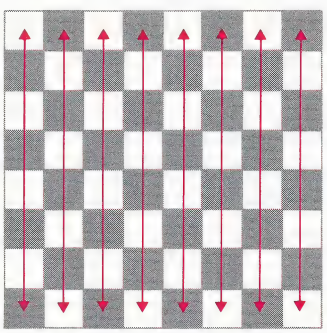
add new information to the head of a list using CONS. When adding to the tail of a list, it is necessary to create a copy of the original list, but with the original final pointer pointing to the new item.

Some versions of LISP have an instruction to do this automatically (APPEND), but it is not as efficient as CONS. If speed is essential, then it is sometimes possible to use an instruction to change the required pointer in the original list, rather than generate a copy — but this sort of 'fiddle' is often frowned upon by purists.

To conclude this series, we'll look at a complete worked example of a well-known problem. The problem is to place eight queens on a normal chess board, such that no queen attacks any other. Obviously, there must be exactly one queen in each column, and one in each row. The problem thus remains to order either the rows or columns so that no two queens intersect along any diagonal.

EIGHT-QUEENS PROBLEM

Imagine that every column on the chess board contains one queen. These will be ordered (1 to 8) according to their places in a list called QUEENS. The queen's row (specified by the value in QUEENS) will be variable, calculated by LISP.



QUEENS (Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8)

The program begins with the list QUEENS being empty. At each stage, a new queen will be placed in row 1 at the beginning of the list. The position of this queen, and possibly others, will be amended (function ALTER) until all of the queens are in safe positions. This new list will become QUEENS, and new queens will be added to the start, and so on.

This continues until the list contains eight queens, at which point they are printed out using PR_BOARD. The initial function, which we will call SOLVE, can thus be written as:

```
(DEFUN SOLVE ()
  (SETQ QUEENS ())
  (LOOP (UNTIL EQ (LENGTH (QUEENS) 8))
    (SETQ QUEENS (ALTER (CONS 1 QUEENS))))
  (PR_BOARD '(QR QN QB Q K KB KN KR) QUEENS))
```

You can see here that SOLVE, with no arguments, initially sets up the empty list QUEENS. The LOOP function then sets QUEENS to the new value of the ALTERed list of a queen in row 1, followed by the previous list. This LOOP is executed UNTIL the LENGTH of QUEENS is EQuivalent to 8. The final list will then be printed.

We have used three new functions here. LENGTH merely returns the number of elements in its argument, and is defined as:

```
(DEFUN LENGTH (L (N))
  (SETQ N 0)
  (LOOP (WHILE L N)
    (SETQ N (ADD1 N))
    (SETQ L (CDR L)) ))
```

The argument N, given in brackets, is Acornsoft LISP's way of showing optional arguments. In this case, N is never passed to the function, and so it just acts as a local variable.

The other function, PR_.BOARD, combines the elements of its two argument lists to produce understandable output. The function is:

```
(DEFUN PR_BOARD (ROWS COLS)
  (COND ((NULL ROWS) T)
    (T (PRINTC (CAR ROWS) (CAR COLS))
      (PR_BOARD (CDR ROWS) (CDR    COLS)) )))
```

Both lists are assumed to be of the same size, so the function terminates if the first condition is true (a list is empty). Otherwise, the first element of each list is printed (with a carriage return), and the function will call itself for the remainder of each list.

The third function, ALTER, is the one that does all the work. It is defined as:

```
(DEFUN ALTER (QUEENS)
  (COND ((EQ (CAR QUEENS) 9)
    (ALTER (CONS (ADD1 (CADR QUEENS))
                 (CDDR QUEENS) )))
    ((SAFE QUEENS) QUEENS)
    (T (ALTER (CONS (ADD1 (CAR QUEENS))
                    (CDR QUEENS) )))))
```

ALTER works by first checking to see whether the most recently placed queen has been moved off the top of the board (row 9). If this is the case, then one or more of the previously placed queens must be moved.

Remember that this queen started in row 1, and will have been checked in all the other rows (by the third condition of ALTER) so it must be the case that it is impossible to place it with the board in the present situation. Consequently, ALTER calls itself with the remainder of the list QUEENS (everything but the present queen).

When doing this, it also increments the row of the last queen. This is called 'backtracking' and is fairly common in computer algorithms of this type. If the second condition evaluates to TRUE, then all the queens on the present board will be SAFE, and the list will be returned. Otherwise, the position of the most recently placed queen will be incremented, and tried again.

The only undefined function here is SAFE, which must check whether all the queens on the present board are safe. Since the position in the list QUEENS defines the column, and no two queens can have the same position in the list, there is no need to check for intersecting columns. This leaves just the rows and diagonals to check. If we assume that we

have a function NOCHECK which returns TRUE if no queen is 'checking' any of the queens preceding it, then SAFE can be defined as:

```
(DEFUN SAFE (QUEENS)
  (COND ((NULL QUEENS) T)
    (T (AND (NOCHECK (CAR QUEENS)
                     (CDR QUEENS) 1)
      (SAFE (CDR QUEENS)) ))))
```

It may be easier to understand this by defining the word SAFE in terms of itself:

'The first queen in the list QUEENS does NOt CHECK any of the queens in the rest of the list, AND all the queens in the rest of the list are SAFE.'

It now only remains to define the function NOCHECK, which should return TRUE if there are no intersections occurring along rows or diagonals. This will look like:

```
(DEFUN NOCHECK (NEW REST COL)
  (COND ((NULL REST) T)
    (T (AND (NOT (EQ NEW (CAR REST)))
      (NOT (EQ COL
        (ABS (DIFFERENCE NEW (CAR REST)))))
      (NOCHECK NEW (CDR REST) (ADD1 COL)) ))))
```

Here, NEW is the recently placed queen, and REST is the list of the other queens on the board. Assuming that there are other queens on the board (NULL REST is FALSE), the AND function returns true if:

1. The present queen's row is not the same as the row of the first queen in the rest of the list.

2. The queens are not on the same diagonal. This is checked by ensuring that the ABSolute row difference, is not the same as the COLumn difference, where COL holds the number of columns separating the present queen from the next queen in the list (initially 1). We have already defined ABS in the previous instalment as:

```
(DEFUN ABS (NUMBER)
  (COND ((MINUSP NUMBER) (MINUS NUMBER))
    (T NUMBER) ))
```

3. 1 and 2 hold true for the present queen and the next queen in the list. This is checked by the function calling itself with everything but the first element of REST and incrementing the COLumn difference.

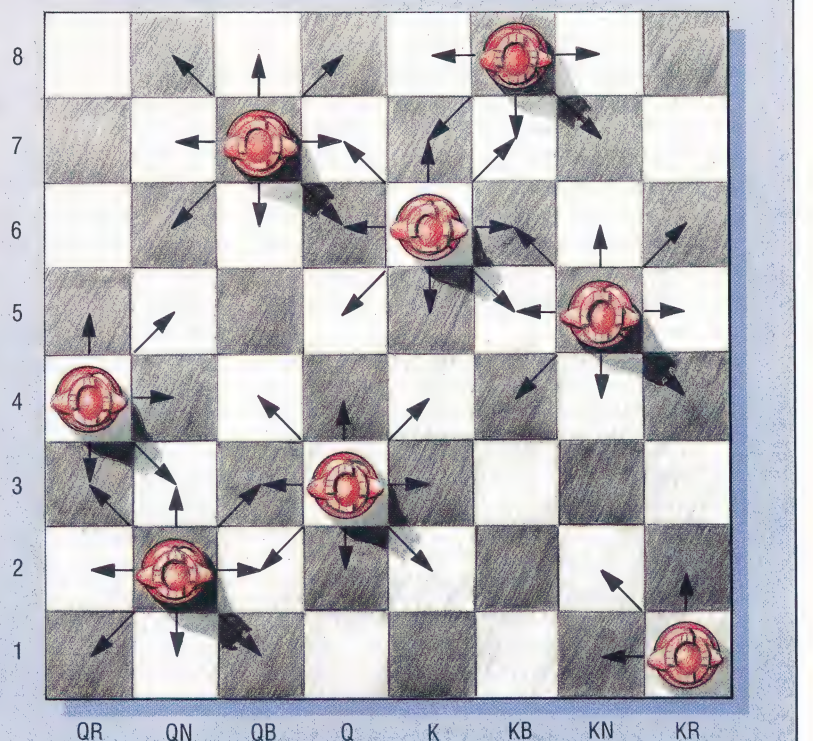
The necessary definitions are now complete, and the illustration shows the solution when it is run.

If you have a LISP implementation, you might like to try these routines. Remember that this has been written using Acornsoft LISP, but shouldn't need much alteration to work on other systems. The backtracking can become quite complex, but you could easily follow it by printing the list QUEENS within the function SOLVE, just before each call to ALTER.

The top-down programming approach to the Eight-Queens problem illustrates how easy it is to define LISP programs on a function-by-function basis. For instance, we used SAFE in ALTER before writing it. This final example should also serve to prove what a versatile language LISP really is, and encourage you to try out your own programs.

A Royal Problem

The Eight-Queens problem provides a perfect example of a puzzle that can be rapidly solved using LISP's facilities for list manipulation and recursive functions. Before you examine closely the solution shown here, you might like to try placing eight queens on a chessboard so that each queen is safe from capture by any of the others. No knowledge of chess is required, apart from the fact that a queen may move any number of squares in a straight line (diagonally, vertically or horizontally) and capture any other piece in its path



DRIVE CONTROL

Having looked at the hook codes used by the Interface 1 in the last instalment (see page 1398), we now consider the functions of those provided for the operation of the Spectrum's Microdrives. Many of these hook codes can be put to great use by the machine code programmer.

Microdrive accesses are carried out either by the use of hook codes (see page 1398) or by calling Interface 1 ROM routines directly. However, the latter procedure is not always a good idea as more than one Interface 1 ROM may be present. So here we will examine the 13 hook codes that have been provided to allow us to use the Microdrive.

Before we do this, let's look at some of the principles behind the use of Microdrives. After a cartridge has been formatted it will be divided into 255 'sectors', which can be regarded as the physical unit of information storage on a cartridge. Each sector consists of a header (data which contains, among other things, the name of the cartridge), a data block, which will eventually hold the name of the file that is using this sector, some other data and a 512-byte data record. The formatting operation also marks for future reference any sectors of the tape in the cartridge that are not usable.

Like all other communication with I/O devices in the Spectrum system, communication between Microdrives and the Spectrum is carried out via a channel. For microdrives, this takes the form of a 595-byte area of memory. Associated with a Microdrive channel is a 32-byte Microdrive map, which is set up by the operating system to tell the various routines which of the sectors are free for use. It will show those sectors that are unusable and those that have already been used for storing other information.

The channel is stored in the area of the Spectrum memory map called the 'Microdrive Channel Area'. Whenever a channel is set up, the memory between the end of the Microdrive Channel Area and STKEND is moved upwards in memory to make space for it. Similarly, when such a channel is no longer needed it is closed and the memory moved down again.

Microdrive channels are set up by the system in two ways: *explicitly*, when a file is required by an OPEN command, or *implicitly*, when the system opens a channel to perform an operation. This is the type of thing done by the Microdrive SAVE command — the channel opening is said to be 'implicit' because it is implicit that a channel be opened to perform the command, even if we will not use it directly. A channel that is opened explicitly must also be manually closed, whereas this is done automatically for an implicit channel.

We provide a diagram showing the structure of a Microdrive channel. You may notice the similarity between the first few bytes of this channel and the channels that we set up to send information to the screen or read data from the keyboard (see page 1317). Most of this information is for interest only; we generally don't have to bother with accessing the Microdrive channel information directly.

The Microdrive map address is held in the channel, and we can get an idea of the available space on a cartridge by examining the map of that cartridge. Indeed, this is how the BASIC CAT function provides you with data about the amount of space remaining on a cartridge. If you want to look at a Microdrive map, then try the following routine (type in NEW first).

```
 10 OPEN #5;"M";1;"testprog"
 20 start=PEEK(23870)+256*PEEK(23871)
 30 FOR I=start TO start+31
 40 LET sector=PEEK(I)
 50 FOR J=1 TO 8
 60 IF sector/2=INT(sector/2) THEN PRINT "0";
 70 IF sector/2<>INT(sector/2) THEN PRINT "1";
 80 LET sector=sector/2:LET sector=INT(sector)
 90 NEXT J
100 PRINT
110 NEXT I
120 CLOSE #5
```

On running this program, a 1 indicates an unusable or used sector, and a 0 indicates a free sector. You might like to consider altering this program to print the amount of free cartridge space to the screen.

With regard to the large, 512-byte section of data in the channel structure diagram, the OS fills this with the data to be written. The data is written to the cartridge if either the buffer is full, or a CLOSE# command (or its hook code equivalent) is executed. In the latter case, the buffer is written to cartridge and that record is marked as an 'End Of File' (EOF) record. It can also be written in a third instance using a hook code, as we'll see later. Under normal conditions, therefore, the cartridge will be written to at 512-byte intervals.

Let's now look at the hook codes used to control the Microdrive.

MICRODRIVE HOOK CODES

- *Hook Code 33:* This starts up the motor in a specified Microdrive unit (a process that is often called 'selecting the drive'). The Microdrives are

numbered from one to eight. The routine can also be used to turn the motors of all Microdrives off. There is one important point to note here, and that is that the routine can return with the interrupts disabled. So, if you use this routine, on returning from it the first thing you should do is issue an EI command.

The routine is simple to use. First, set the A register to hold the number of the drive that you want to turn on. Then use RST #8 to call the hook code. Placing the value 0 in the A register will turn off the drives. Alternatively, the Microdrives can be turned off by generating an error in BASIC. Errors can be generated by this hook code in two ways: if you try to turn on a drive that is not connected, and if you try to turn on a drive without a formatted cartridge in it.

● *Hook Code 34:* This very useful call enables us to open a file on a cartridge. We'll look at the setting up needed for this call shortly. However, once called, the routine will check the drive for a file of the name that you've given. If one is found, then the file will be opened for *reading*. If not, then it will be opened for *writing*. This is reflected in the state of the Read/Write flag of the channel that this call sets up: bit 0 will be 0 for a read file and 1 for a write file.

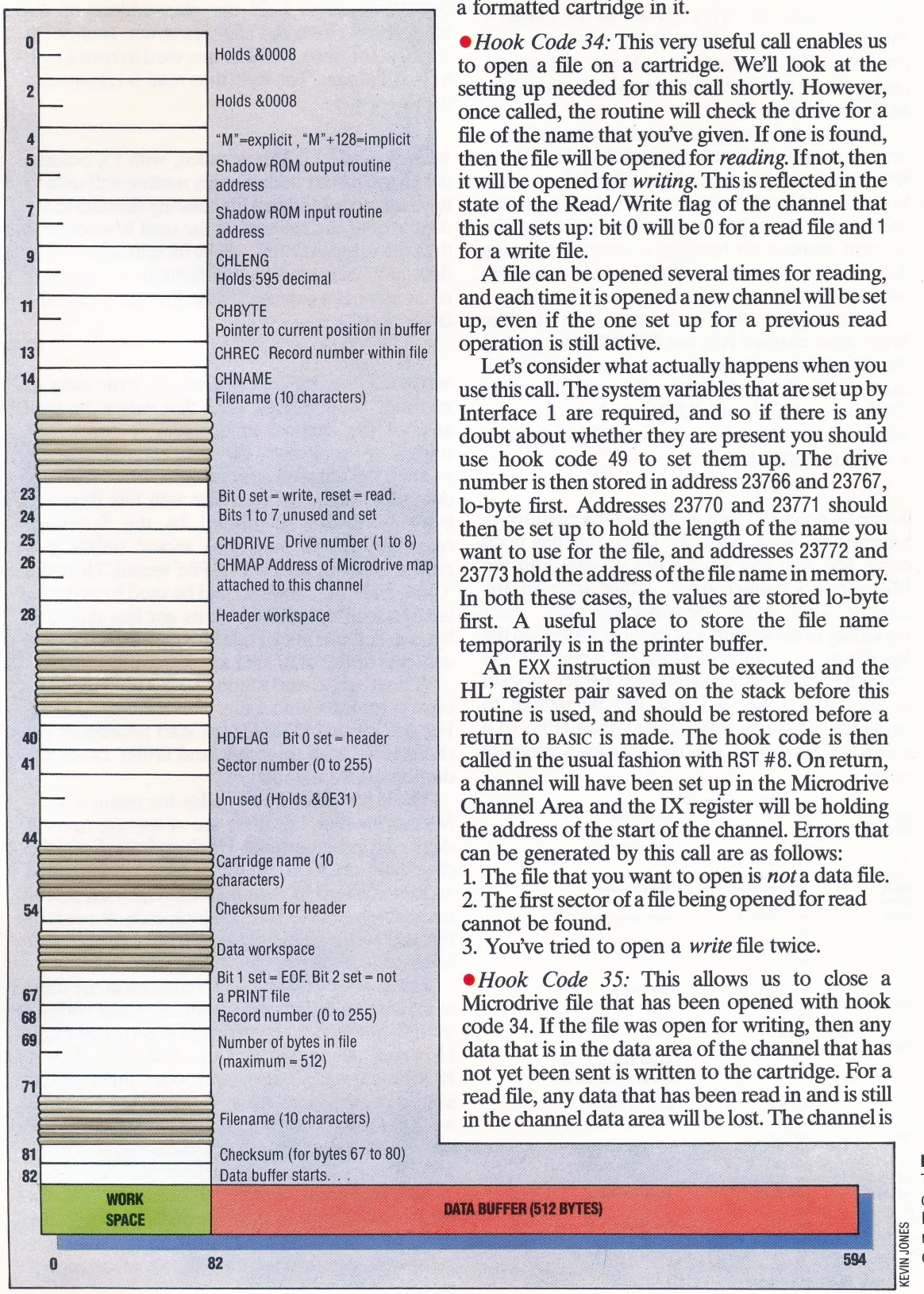
A file can be opened several times for reading, and each time it is opened a new channel will be set up, even if the one set up for a previous read operation is still active.

Let's consider what actually happens when you use this call. The system variables that are set up by Interface 1 are required, and so if there is any doubt about whether they are present you should use hook code 49 to set them up. The drive number is then stored in address 23766 and 23767, lo-byte first. Addresses 23770 and 23771 should then be set up to hold the length of the name you want to use for the file, and addresses 23772 and 23773 hold the address of the file name in memory. In both these cases, the values are stored lo-byte first. A useful place to store the file name temporarily is in the printer buffer.

An EXX instruction must be executed and the HL' register pair saved on the stack before this routine is used, and should be restored before a return to BASIC is made. The hook code is then called in the usual fashion with RST #8. On return, a channel will have been set up in the Microdrive Channel Area and the IX register will be holding the address of the start of the channel. Errors that can be generated by this call are as follows:

1. The file that you want to open is *not* a data file.
2. The first sector of a file being opened for read cannot be found.
3. You've tried to open a *write* file twice.

● *Hook Code 35:* This allows us to close a Microdrive file that has been opened with hook code 34. If the file was open for writing, then any data that is in the data area of the channel that has not yet been sent is written to the cartridge. For a read file, any data that has been read in and is still in the channel data area will be lost. The channel is



KEVIN JONES

Microdrive Channel Structure
The Microdrive channel occupies 595 bytes of memory. Its location is returned in the IX register after opening a file on cartridge using hook code 34

then reclaimed by moving the area of memory up to STKEND down over it. The routine is called with the IX register holding the address of the start of the channel we have specified. Again, it's a good idea to preserve the alternative HL' register pair before use. An error is generated if there is insufficient room on the cartridge to close the file.

● *Hook Code 36:* This allows us to delete a Microdrive file, whether it's a data file or a program. The file name, its address and length, and the drive number are all set up in the same way as for hook code 34.

Now that we've seen how we can open and close files, it would obviously be useful to be able to write and read data from those files! It is possible to attach a Microdrive channel to a stream, but the easiest way is to make the Microdrive channel the 'current' channel for reading or writing (replacing the usual keyboard and screen streams respectively).

The simplest way to write data to the Microdrive channel is to use RST #10, after setting the Microdrive channel as the current output channel. We do this by setting the system variable CURCHL, at 23633 and 23634, to hold the start address of the Microdrive channel. This is done by a single command:

```
LD          (23633),IX
```

where IX holds the start address of the channel, as returned by hook code 34. Once this has been done, RST #10 will write bytes of data into the Microdrive Channel Area, rather than to the screen. This again demonstrates the sheer versatility of the stream and channel system on the Spectrum — a much neglected aspect of the computer's OS. As we mentioned earlier, as soon as the data buffer is full it will be written to cartridge. Thus, to write a file called FRED to a cartridge, we could use the following piece of code:

```
                ;write file "FRED" to microdrive
3E01            ld   a,1                ;drive number
32D65C          ld   (23766),a          ;store drive number
AF              xor  a                  ;zero a register
32D75C          ld   (23767),a
212B29          ld   hl,name            ;get name in hl
22DC5C          ld   (23772),hl         ;store filename
210400          ld   hl,4               ;4=name length
22DA5C          ld   (23770),hl         ;store name length
D9              exx
E5              push hl                 ;preserve hl'
D9              exx
CF              rst  #8
22              defb 34                 ;hook code 34
DD22515C        ld   (23633),ix         ;CURCHL=drive channel
06FF            ld   b,255              ;255 bytes infile
C5          loop: push bc               ;preserve counter
3E00            ld   a,CHAR             ;user would need to
                                        ;insert routine to get
                                        ;character from wherever
                                        ;into a register
D7              rst  #10                ;write to channel
C1              pop  bc                 ;restore counter
10F6            djnz loop
CF              rst  #8
23              defb 35                 ;close file
3E02            ld   a,2
CD0116          call #1601              ;output to screen
D9              exx
E1              pop  hl                 ;restore HL
D9              exx
FB              ei                      ;reenable interrupts
C9              ret
46524544    name: defm "FRED"           ;filename
```

Obviously, if you open a couple of files it's necessary to save the channel start addresses of each channel in some safe area of memory (so that they can be used to close the files, and so on). Our routine generates a file of 255 characters to the cartridge.

Reading the data from such a file is also straightforward; the file is opened as before, and CURCHL made to hold the start address of the Microdrive channel. The Spectrum routine at #15E6 in the main ROM is then used to read a byte from a channel. The byte thus read is returned in the A register.

● *Hook Code 37:* When called, with IX holding the channel start address, this routine will read in the next record within a file, placing the data in the data area of the channel. If the read is successful, then the value of CHREC will be incremented. Once data has been read in this fashion, it obviously replaces what was previously in the data area of the channel.

● *Hook Code 38:* It is possible to write data to cartridge using a hook code that writes the data area of the channel in question to tape. This routine — hook code 38 — is entered with IX holding the channel start address. Note that any data area written to cartridge with less then 512 bytes in it can be treated by the Spectrum operating system as a 'free sector' unless it is marked as being an 'End Of File' record. Thus, the 'Close File' hook code should be used to write the last block of data to tape if its got less than 512 bytes in it. Once hook code 38 is called, the current contents of the data area are written to file.

A final useful read routine allows us to read in a certain record within a file. This is hook code 39. For this routine IX holds the start address of the channel we wish to access, and CHREC holds the desired record number.

There are other hook codes for manipulating Microdrive files, but these are of less use to us in day to day programming. Hook code 40 allows the data held on a particular Microdrive cartridge *sector* to be read in. Again, CHREC holds the sector number we wish to access. Hook code 41 reads in the next sector from the Microdrive cartridge, and hook code 42 writes data to the next free sector.

The final two hook codes we'll look at are used to manipulate Microdrive channels. Hook code 43 should produce a channel and a cartridge map. However, in the first version of the Interface 1 ROM there was a bug in this routine that resulted in it doing the same job as hook code 34. It does work in later versions of the ROM, though. Hook code 44 deletes a channel, and the associated map, which has its start address held in the IX register.

As a final point, care should be taken when you are experimenting with hook codes, because the write-protect system used by the Microdrive is for *software* only. A crash can therefore corrupt a cartridge!

DATABASE

Here, courtesy of Commodore Business Machines, we publish another instalment of the Commodore 64's memory map

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
SYREG	030E	782	Storage for 6502 .Y Register
SPREG	030F	783	Storage for 6502 .SP Register
USRPOK	0310	784	USR Function Jump Instr (4C)
USRADD	0311–0312	785–786	USR Address Low Byte/ High Byte
	0313	787	Unused
CINV	0314–0315	788–789	Vector: Hardware IRQ Interrupt
CBINV	0316–0317	790–791	Vector: BRK Instr. Interrupt
NMINV	0318–0319	792–793	Vector: Non-Maskable Interrupt
IOPEN	031A–031B	794–795	KERNAL OPEN Routine Vector
ICLOSE	031C–031D	796–797	KERNAL CLOSE Routine Vector
ICHKIN	031E–031F	798–799	KERNAL CHKIN Routine Vector
ICKOUT	0320–0321	800–801	KERNAL CHKOUT Routine Vector
ICLRCH	0322–0323	802–803	KERNAL CLRCHN Routine Vector
IBASIN	0324–0325	804–805	KERNAL CHRIN Routine Vector
IBSOUT	0326–0327	806–807	KERNAL CHROUT Routine Vector
ISTOP	0328–0329	808–809	KERNAL STOP Routine Vector
IGETIN	032A–032B	810–811	KERNAL GETIN Routine Vector
ICLALL	032C–032D	812–813	KERNAL CLALL Routine Vector
USRCMD	032E–032F	814–815	User-Defined Vector
ILOAD	0330–0331	816–817	KERNAL LOAD Routine Vector

© 1983 JULICH, J.—SHERIDAN COLLEGE